# IDA

# State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016

E. Kenneth Hong Fong, *Project Leader*

David A. Wheeler
Amy E. Henninger

**IDA** *The Institute for Defense Analyses is a non-profit corporation that operates three federally funded research and development centers to provide objective analyses of national security issues, particularly those requiring scientific and technical expertise, and conduct related research on other national challenges.*

INSTITUTE FOR DEFENSE ANALYSES

# State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016

E. Kenneth Hong Fong, *Project Leader*

David A. Wheeler
Amy E. Henninger

# Executive Summary

Nearly all modern systems depend on software. It may be embedded within the system, delivering capability; used in the design and development of the system; or used to manage and control the system, possibly through other systems. Software may be acquired as a commercial off-the-shelf component, custom developed for the system, or embedded within subcomponents by their manufacturers. Modern systems often perform the majority of their functions through software and can easily include millions of lines of software code.

Although functionality is often created through software, this software can also introduce risks. Unintentional or intentionally inserted vulnerabilities (including previously known vulnerabilities) can provide adversaries with various avenues to reduce system effectiveness, render systems useless, or even turn our systems against us. Department of Defense (DoD) software, in particular, is subject to attack. Analyzing DoD software to identify and remove weaknesses is a critical program protection countermeasure. Unfortunately, it can be difficult to determine what types of tools and techniques exist for analyzing software, and where their use is appropriate.

The purpose of this paper is to assist Department of Defense (DoD) program managers (PM), and their staffs, in making effective software assurance (SwA) and software supply chain risk management (SCRM) decisions, particularly when they are developing their program protection plan (PPP). A secondary purpose is to inform DoD policymakers who are developing software policies.

This paper defines and describes the following overall process for selecting and using appropriate analysis tools and techniques for evaluating software:

1. *Select technical objectives based on context.* This paper identifies a set of 10 major technical objectives and subdivides them further into up to 3 more levels of progressively more detailed objectives. For example, the major technical objective "counter unintentional-'like' weaknesses" is subdivided into a second level of 12 sub-categories, and some of these second-level objectives are subdivided still further. This multi-stage breakdown of technical objectives is captured as the table rows in Appendix E, Software State-of-the-Art Resources (SOAR) Matrix.

2. *Select tool/technique types to address those technical objectives.* This paper identifies 59 types of tools and techniques available for analyzing software. The

supporting "Software SOAR Matrix" provides a detailed mapping between these tool/technique types and the technical objectives, to help readers identify and select the types of tools and techniques to meet the technical objectives.

3. *Select tools/techniques.* This paper identifies, in some cases, where additional information is available to help the selection process.

4. *Summarize selection as part of a Program Protection Plan.* This paper provides guidance on how to summarize the information derived from the selection of tool/technique types, and later the planned use of the tools/techniques, into a PPP.

5. *Apply the tools/techniques and report the results.* Here the selected tools and techniques are applied, including the selection, modification, or risk mitigation of software based on tool/technique results. Reports are provided to support oversight and governance.

Vignettes in Section 8 provide examples of this process. This paper also describes some key gaps that were identified in the course of this study, including difficulties in finding unknown malicious code, obtaining quantitative data, analyzing binaries without debug symbols, and obtaining assurance of development tools. Additional challenges were found in the mobile environment; examples include lack of maturity in many tools, expectations of time constraints that preclude in-depth analysis, and widespread use of a Software-as-a-Service (SaaS) model that limits data availability and application to DoD systems. These would be plausible areas to consider as part of a research program.

Appendices provide additional detail, including more information on each type of tool and technique. Appendix D, for example, describes how we believe analysis should be continuously applied and integrated into the entire software lifecycle, creating a feedback loop for better-informed risk management decisions.

The information provided here was gathered from a variety of sources, including many interviews of subject matter experts. These experts identified a number of key topics, some of which are also captured in this paper.

This paper extends the earlier editions, in response to technology advancements and reviewer feedback. Appendix G presents in more detail the changes made in this update.

Software analysis is a large and dynamic field, and this paper represents one step in capturing and organizing a wide range of diverse information. We hope that this material will continue to be refined through feedback from the larger community.

# Contents

# 1.    Introduction

The purpose of this paper, "State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation," is to assist Department of Defense (DoD) program managers (PM), and their staffs, in making effective software assurance[1] (SwA) and software supply chain risk management[2] (SCRM) decisions, particularly when they are developing their program protection plan (PPP).  A secondary purpose is to inform DoD policymakers who are developing software-related policies.

Nearly all modern systems depend on software.  This software has become increasingly large and complex, including many subsystems that are composed of even more subsystems.  Some software is custom-developed, but a great deal is off-the-shelf (OTS).  This OTS software may be from the Federal Government, other governments, the technical community, or the marketplace as either open or proprietary commercial (commercial off-the-shelf (COTS)) products.  Software may be provided as a discrete end-item, or embedded within larger assemblies or packages.

Although software creates functionality, it also poses risks.  Unintentional and intentional vulnerabilities (either originally included or inserted later) can provide adversaries various avenues through which to reduce system effectiveness, render systems useless, or even turn our systems against us.  The DoD, in particular, is under constant attack.

Various methods exist to evaluate risk.  For the purposes of this paper, they can be divided into methods to evaluate *people*, *processes*, and *products*.

- People-related methods are used to evaluate risk by examining the individuals and their organizations that supply goods and services.  For example, all-source intelligence can be used to look for evidence that the suppliers are intentionally providing vulnerable products, which suppliers are being targeted by

---

[1]   Software assurance (SwA) is defined in [CNSS2015] as "The level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software throughout the lifecycle."

[2]   Supply chain risk management is defined in [CNSS2015] as "A systematic process for managing supply chain risk by identifying susceptibilities, vulnerabilities, and threats throughout the supply chain and developing mitigation strategies to combat those threats whether presented by the supplier, the supplies product and its subcomponents, or the supply chain (e.g., initial production, packaging, handling, storage, transport, mission operation, and disposal."

adversaries, and which suppliers are more likely to be producing unintentionally vulnerable products (e.g., based upon reports of known exploitations).

- Process-related methods evaluate risk by examining the processes used to develop and sustain the goods and services, with the intention of discerning likely weaknesses and characteristics of the software produced.

- Product-related methods are used to evaluate risk by examining the goods and services themselves.

Methods that are used to examine people and processes have important limitations. Both people- and process-related evaluation methods are indirect methods for evaluating the actual software produced, yet it is the produced software that actually matters. Additionally, it is often difficult to identify suppliers (especially at lower tiers), and even when they are identified, it is often difficult to evaluate risk based on the information available.

Thus, it is valuable to directly evaluate software products that may be used. This research, therefore, focuses on tools and techniques for directly evaluating software as a product. Happily, many types of tools and techniques are available for directly evaluating software. Unfortunately, it can be difficult to determine what types of tools and techniques are relevant and when their use is appropriate.

This paper addresses this difficulty, by identifying *types* of tools and techniques available for evaluating software, as well as the *technical objectives* those tools and techniques can meet. This paper focuses on evaluating software for unintentional and intentional vulnerabilities, but some tools and techniques also address other issues.

This paper discusses types of *both* tools and techniques. Tools are primarily automated systems for evaluating software, although there is typically some manual effort in their use (e.g., to configure, review results, and apply mitigations based on those results). Techniques are primarily manual (human) approaches for evaluating software, although there are typically some automated systems for aiding the manual approach (e.g., for tracking progress and exchanging data). A potential advantage of tools is scalability; manual approaches can be too costly or time-consuming for large software systems. However, techniques can have significant advantages in terms of their ability to handle context and to focus on what is important.

The tools, suppliers, and organizations named in this paper are used as representative examples and our lists are not complete. Products and organizations are constantly evolving, and features are added on a frequent basis. Further, no endorsement of particular tools, suppliers, or organizations is intended.

In this paper, the software being evaluated is called the target of evaluation (TOE). As discussed later in the paper, the context in which the software will be used is key; the

kind of software, and its criticality or mission, are key determinants in deciding whether or not a particular tool is appropriate. Software evaluation is a challenging problem; addressing a common set of technical objectives often requires a suite of tools and techniques.

By itself, this paper does not define an overall strategy for acquiring secure software, but it and the types of tools and techniques we have identified can support such an overall strategy.

The information provided here was gathered from a variety of sources, including many interviews of various experts. We are especially grateful to the interviewees for their time, and we are also grateful to the Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics (OUSD(AT&L)), DoD Chief Information Officer (CIO), and National Security Agency (NSA) for co-sponsoring this work.

Chapter 2 provides a brief background. Chapter 3 describes a possible overall process for selecting appropriate analysis tool/technique types. This process involves identifying technical objectives (described in Chapter 4), and selecting tool/technique types (described in Chapter 5) to address those technical objectives based on the software context (described in Chapter 6). Specific tools and techniques are then selected. The plans for a given project, and eventually the results of selecting and applying these tools and techniques, should be summarized; Chapter 7 describes how to capture this summary information in a DoD PPP. Chapter 9 provides some vignettes that serve as examples of the first few steps. Chapter 10 lists key gaps identified in the course of our investigations. Finally, Chapter 11 presents this paper's conclusions. The appendices provide additional detail. Appendix C, in particular, provides detailed fact sheets on each tool/technique type. Appendix E (provided as a separate electronic file) summarizes how well the tool/technique types address different technical objectives. Since it is difficult to quantify this relationship, entries in this appendix represent the authors' summary of information derived from a variety of sources, and are not the result of analysis from any one standard testing regime. Appendix F discusses the mobile environment.

Appendix G describes the major changes made in various revisions of this document. Some changes were inspired by changes in technology, but many others were inspired by feedback from our many reviewers, including those from IDA, MITRE, and the Software Engineering Institute (SEI). We gratefully acknowledge our reviewers' feedback.

# 2.   Background

Other works have summarized the landscape of software assurance (SwA) and/or software supply chain risk management (SCRM).  In particular, [Goertzel 2007] presents an overview; however, it focuses on describing the "state of the art" rather than practical application.  A previous effort, termed "SOAR-Lite Phase I" [Wheeler 2012], identified current research in software assurance and SCRM, much of which was insufficiently mature for regular use.

In contrast, this SOAR focuses on the practical application of leading-edge but sufficiently mature technology.  The initial version was released in 2013 and was updated in 2014.  This paper updates previous versions as described in Appendix G.

Our approach when developing the initial 2013 version was to first gather relevant information, and then organize it in a coherent fashion.  We gathered information from a wide variety of interviews with members of the government community (both within and outside the DoD military Services), as well as various vendors and suppliers.  We also reviewed many written materials and attended a variety of relevant meetings and conferences.  Appendix A provides more information on the people we interviewed and the resources we used.  Later versions were developed based on feedback.

This research is part of a larger ongoing effort by the DoD to improve SwA and supply chain assurance.  Other documents that describe or enable this process include:

- Deputy Assistant Secretary of Defense – Systems Engineering, Program Protection Plan Outline & Guidance. [DASD(SE) 2011];

- DoD Instruction 5200.44, Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN) [DoDI 5200.44];

- Defense Acquisition Guidebook (DAG), particularly chapter 13 [DAG];

- Public law 112–239—Jan. 2, 2013, National Defense Authorization Act (NDAA) for fiscal year 2013, particularly section 933 [Congress 2013].

It is important to consider assurance throughout the software development lifecycle (SDLC).  The SDLC includes development (requirements, design, implementation, and test), deployment, operations, sustainment, and disposal.  We focus on the DoD lifecycle as described in DoD Instruction 5000.02 [DoDI 5000.02].

This paper focuses on software.  DoD Federal Acquisition Regulation (FAR) Supplement (DFARS), section 252.227-7014 ("Rights in Noncommercial Computer

Software and Noncommercial Computer Software Documentation"), defines computer software as "computer programs, source code, source code listings, object code listings, design details, algorithms, processes, flow charts, formulae, and related material that would enable the software to be reproduced, recreated, or recompiled." It also defines a computer program as "a set of instructions, rules, or routines, recorded in a form that is capable of causing a computer to perform a specific operation or series of operations." For this paper we consider a computer program to be data that executes on a central processing unit (CPU) or graphics processing unit (GPU), including the CPUs in the controllers of products such as printers and cars. The term "software" includes firmware, operating systems, and middleware, as well as applications. The term "software" does not include computer hardware. We also do not include as software the configuration data that programs a Field Programmable Gate Array (FPGA) (i.e., the bitstream), including data derived from data written in Verilog or VHDL[3]. The software tools for designing, developing, and fabricating hardware components are software, and they are critically important when using encoded microelectronics. These tools include those for developing (not using) FPGAs and Application Specific Integrated Circuits (ASIC). However, we do not specially consider these tools here. Software included in the application supported by the FPGA should be analyzed as described in this paper.

A few key basics about software and software development are helpful in understanding this paper. "Source code" is the set of computer instructions in a human-readable computer language that is written and maintained by software developers. In many situations source code is translated into a "bytecode" or "binary" (using a program or device called a compiler). Binaries are representations that can be directly executed by the computer, while bytecode is an intermediate representation that is executed by some other program. It is important to know the distinction between source code, bytecode, and binaries, because some OTS software suppliers will only provide bytecode or binaries, yet some analysis tools require source code to perform their analysis. Commercial OTS (COTS) suppliers provide software under a variety of licenses. Some COTS software is licensed as open source software (OSS). OSS is software for which the human-readable source code is "available for use, study, reuse, modification, enhancement, and redistribution by the users of that software" [DoD 2009]. COTS software that is not OSS is often referred to as "proprietary" or "closed" software; the source code for such software is often not available or only available at additional cost.

In many cases software is developed by combining a set of software components, which are in turn made from other software components. These different components are themselves often provided by different suppliers. Libraries and frameworks are types of software components that are designed to be reused by other components.

---

[3] VHDL stands for Very-high-speed integrated circuit (VHSIC) Hardware Description Language.

A "regression test suite" is a set of tests that can be automatically re-invoked. These are used to ensure that a change in software does not cause some other function to malfunction.

Most tools and techniques are subject to the problems of false positives and/or false negatives. A "false positive" is a report that is invalid; e.g., if a tool is intended to report vulnerabilities, a false positive is a report of a situation that is not actually a vulnerability as a vulnerability. A "false negative" is a failure to report a situation, e.g., a false negative occurs when a vulnerability is not reported but should be reported.

Some tools are "sound." The National Institute of Standards and Technology (NIST) defines a "sound tool" as follows, a definition it calls the "Ockham Sound Analysis Criteria" [NIST Ockham]:

- "A *site* is a location in code where a weakness might occur."

- "A *finding* is a definitive report about a site. In other words, that the site has a specific weakness (is buggy) or that the site does not have a specific weakness (is not buggy)."

- "A *sound* tool is a tool for which every finding is correct. The tool need not produce a finding for every site; that is *completeness*."

Some environments, such as Android, support "permission manifests." In such systems, each application includes a permission manifest, which is a static set of permissions that the application claims to require. Note that "permissions" in this context are the privileges granted to an application, not the permissions set on objects such as files or memory. Some tools examine these permission manifests, either by themselves or through comparison with other information. Examples of such permissions (privileges) include location information (at a coarse or fine level), use of certain sensors (e.g., microphone access), and network access.

# 3. Overall Process for Selecting and Reporting Results from Appropriate Tools and Techniques

We have identified many different types of analysis tools and techniques. Selecting among these tool and technique types depends on the software context and technical objectives. This chapter describes the overall process for selecting types of tools and techniques, selecting specific tools and techniques, and reporting their results.

## A. General Approach

Our proposed approach for selecting various tools and techniques, and developing reports using them, is to first identify the software components in a target of evaluation (TOE) and determine each software component's context of use (as described in section 6.A).

Then, for each software component context of use:

1. Identify technical objectives based on context. Technical objectives are discussed in Chapter 4. Applying this information to select specific technical objectives is further discussed in section 8.A.

2. Select tool/technique types needed to address the technical objectives, using the matrix discussed below (in section 3.B) and presented in Appendix E. Tool/technique types are discussed in Chapter 5 and the fact sheets in the appendices. Applying this information to select specific tool/technique types is further discussed in section 8.B.

3. Select specific tools (see guidance in Chapter 5 and context as described in Chapter 6).

4. Summarize selection (write down your plan), which may be part of a larger report. In the DoD, this would be part of the Program Protection Plan (PPP) (see Chapter 7).

5. Apply the analysis tools, use their results, and report appropriately. Here the selected tools and techniques are applied, including the selection, modification, or risk mitigation of software based on tool/technique results, and reports are provided to those with oversight authority.

## B. Matrix to Help Select Tool/Technique Types to Address Technical Objectives

Since different tool/technique types are better at addressing different technical objectives, we suggest ensuring that the set of tool/techniques selected adequately cover the intended technical objectives. One way to do this is to use a matrix we have developed that specifies the technical objectives met, to some degree, by various tool/technique types. This section describes key parts of the matrix.

Figure 3-1 shows the general outline of the matrix. The full matrix is in Appendix E, Software SOAR Matrix. On the left-hand side is the set of technical objectives (discussed in Chapter 4). Many technical objectives are subdivided further into lower-level objectives, and these subdivisions are shown as additional columns. A "need indicator" helps identify when certain technical objectives may be important, e.g., countering buffer overflows may be important in programs written in the C, C++, and Objective-C languages, yet are often irrelevant otherwise. The next set of columns identifies various tool/techniques types (per Chapter 5); these are arbitrarily numbered (as 1, 2, and so on) in this general outline to simplify this diagram. The tool/technique types are grouped into three larger categories: static, dynamic, and hybrid.

| Technical Objective | Lower-level technical objective | Need indicator | Tool/technique type | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Static | | | Dynamic | | | Hybrid | |
| | | | 1 | 2 | … | 21 | 22 | … | 31 | … |
| **Design & code quality** | | | | | | | | | | |
| **Counter unintentional-like known vulnerabilities** | … | | | √ | | | | | | |
| **Authentication & access control** | Authentication | | | | | | | | | |
| | … | | | | | | √ | | | |
| **Counter unintentional-like weaknesses** | Buffer handling | C/C++/ Objective-C | | | | | | | | |
| … | … | | | | | | | | | |

**Figure 3-1. Matrix Outline**

The cells in the matrix that connect technical objectives with tool/technology types indicate the applicability of the tool/technology type for addressing that technical objective, as determined by the authors. These indicators are:

- ⊘ = Completely addresses this objective. This indicator is, unfortunately, rarely used. These cells are shaded green.

- ☑ = Can be a highly cost-effective measure to address this objective; investigate further.  These cells are shaded yellow.

- ◐ = Can be cost-effective for partial coverage of this objective.  These cells are shaded orange.

- ▬ = Not identified as being typically applied for this objective.

Cell entries represent the authors' qualitative summary of information provided by a variety of sources.  We examined the information we had gathered (including interviews and documents), aggregated the whole set, then debated until the authors agreed on a rating for each cell (each cell pairs a tool/technique type with a technical objective).[4] Cell values represent the best expected value for that tool/technique type; a specific tool that implements a given type will not necessarily produce this best expected value for a given technical objective and context.  Tools are often designed to only process a particular kind of application (e.g., a web application) and/or set of programming languages (e.g., C, Java, C#, or Python).  In addition, most tools use rulesets and/or heuristics; different rulesets and heuristics can produce significantly different results.  In some cases additional commentary was attached to cells.  Cell entries are not the result of analysis from any standard testing regime, because it is currently difficult to quantify this relationship and few standard testing regimes are available.   The NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project and NSA Center for Assured Software (CAS) have developed and are running test suites for a few specific tool/technique types, but attempting to expand such a process to all the tool/technique types and technical objects that we address would have exceeded the resources available. We expect this relationship indication can be improved over time with contributions from the community, experience, and advances in measurement.  Still, we believe that there is value in providing subject matter expert guidance for typical cases.

The full matrix (but not the excerpt matrix outline above) includes an additional column, titled "best applicability," immediately to the left of the first tool/technique type column.  The "best applicability" column shows the best applicability value for each row, e.g., if there is some tool/technique type that completely addresses the objective in a given row, this column will have the indicator for "completely addresses this objective." This column answers the question, "what would happen if all of tools/techniques listed were used?"   This column reveals that only a very few technical objectives are

---

[4]  Versions previous to 2016 were written by David A. Wheeler and Rama S. Moorthy. The additions for the 2016 version were developed by authors David A. Wheeler and Amy E. Henninger, based on the previous work.  We thank Rama S. Moorthy for her work on the previous versions.

completely addressed. This column also reveals that there is especially poor coverage for countering some kinds of intentional-"like"/malicious logic.

The full matrix (but not the excerpt matrix outline above) also includes an estimated "cost to implement" and "subject matter expert (SME) expertise" entry for each tool/technique type. These entries provide a rough *qualitative* estimate for using the tool/technique for a given project, as estimated by the authors. These qualitative estimates are primarily comparisons between examples of different tool/technique types, because costs and necessary expertise for a given situation will widely vary depending on many factors (such as the size of the software). In particular, a tool may cost somewhat more if it is a higher-end tool for a given type or if it is licensed for an especially large project. In some cases it is difficult to give even a qualitative estimate, so in those cases a range is given. The purpose of these estimates is not to provide detailed cost or expertise estimations, because actual values can vary considerably. Instead, their purpose is to identify qualitative differences between different tool/technique types. For example, in practically all cases, the tool/technique type "full manual source code review" will cost far more than using "warning flags" or "traditional virus detection," regardless of the program. It is these *qualitative* differences that "cost to implement" and "SME expertise" are designed to reflect. Thus, while a specific tool may vary in cost or required expertise compared to the entries given here, these qualitative values provide useful indicators when comparing different tool/technique types.

The "cost to implement" entry has the following codes for each tool/technique type:

- $: The tool/technique has costs similar to the tool/technique type's "warning flags" (when initiated before code development) and "traditional virus detection." It is already included in existing toolsets, free, or can often be acquired for less than $10K, and the annual cost to operate and remediate (at least in certain circumstances) is similarly low. Note that costs can vary considerably by circumstance. For example, the "warning flags" approach is often low-cost if warning flags are enabled *before* writing software, since developers can typically quickly remediate and learn to avoid the circumstances that trigger warnings while they are initially developing software. In contrast, enabling warning flags can be much more expensive if applied to already-written software, since there is often a large amount to remediate, and each remediation typically requires a more extended follow-on analysis.

- $$: The tool/technique acquisition has costs similar to the tool/technique type "source code weakness analyzers." This type often costs $10K-$50K to initially acquire, including the tools themselves. In addition, there are nontrivial costs to analyze tool results and then to implement mitigations.

- $$$: The tool/technique acquisition has costs similar to the tool/technique type "context-configured source code weakness analyzer." This tool/technique type is often more expensive than "source code weakness analyzers" because it configures a source code weakness analyzer specifically for the product being evaluated (e.g., by adding many additional rules). This type often costs $50-$200K to initially acquire, including the tools themselves. In addition, there are nontrivial costs to analyze tool results and then to implement mitigations.

- $$$$: The tool/technique has very large costs similar to the tool/technique type "full manual source code review"; these costs are typically due to a large amount of expert manual labor. This type often costs over $200K for acquisition and application to a medium or larger-sized project.

Note that many tools in practice require some sort of annual maintenance fee, as well as general administrative maintenance to continue working; often these costs are some percentage of the initial purchase price.

The "subject matter expert (SME) expertise" entry has the following codes for each tool/technique type:

- E: The training time is similar to the tool/technique type "enabling warning flags." This is often 3 weeks or less for a typical developer or system analyst.

- EE: The training time is similar to "source code weakness analyzer" and "framework-based fuzzer." This often requires 1 to 4 months for a typical developer or system analyst to become proficient (though it may be much less to get started). Experts are not too difficult to find or train, but note that they tend to have 5 or more years' experience in development/analysis.

- EEE: The training time is similar to "focused manual spot check - Focused manual analysis of source." This often requires 4 to 9 months for a typical developer or system analyst to become proficient. Experts can be difficult to find, and may be quite senior (thus costly).

- EEEE: The training time is similar the tool/technique types "debugging" (for the purpose of thoroughly analyzing previously-unseen software for software assurance) and "formal methods." This often requires more than 9 months for a typical developer or system analyst to become proficient, and that proficiency may be tied to a particular product being examined. Experts are relatively rare, and may be quite senior (thus costly).

## C.  Using the Matrix

To use the matrix, identify a set of technical objectives, and then identify a set of tool/technique types that might help meet those objectives. Then investigate those

tool/technique types for applicability on that particular component/system (the fact sheets in Appendix C, as well as the estimates of cost and required expertise, can help do this). Then select tool/technique types to cover the technical objectives and verify that they cover the technical objective.

Using multiple types of tools and techniques for each technical objective would typically provide better coverage, since different types provide different kinds of information. Doing so often costs more, since more tools are being applied, and integrating multiple data results from dissimilar sources often requires additional investment and knowledge.

The National Defense Authorization Act for Fiscal Year 2014 [NDAA 2014] section 937 requires the establishment of a "joint federation of capabilities to support the trusted defense system needs of the Department of Defense," including a charter to set forth the "the requirements for the federation to procure, manage, and distribute enterprise licenses for automated software vulnerability analysis tools." This matrix may help members of this federation perform these tasks by helping them identify types of tools and techniques that could be especially helpful in different circumstances.

The following chapters describe in more detail proposed technical objectives, tool/techniques types, and software contexts, so that this matrix can be effectively used.

# 4.    Technical Objectives

Different types of tools and techniques are better for different purposes.  Thus, it is important to identify the various purposes for using different types of tools and techniques, so that the most appropriate types can be selected.  In this paper, these purposes are called *technical objectives*.

Chronologically the software context should be determined first, and then the set of technical objectives for that context should be determined.  However, it is more challenging to explain how to select technical objectives for a given context without first explaining technical objectives.  Therefore, this paper explains technical objectives first; Chapter 6 then discusses selecting technical objectives for a given context.

The following sections describe how this set of technical objectives was developed, followed by a summary of the top levels of the technical objectives.

## A.   Technical Objectives' Development Approach

It is common for security issues to be categorized as being related to confidentiality, integrity, and availability; the DoD also separately considers authentication and non-repudiation [DoDI 8500.01].  However, since a vulnerability can cause problems in all of those areas, these categorizations are too general to support narrowing the selection of appropriate tool/technique types.

Even at a more detailed level, there is no universally accepted set of categories for technical objectives.  The Common Weakness Enumeration (CWE) identifies a very large set of common weaknesses in software that may lead to vulnerabilities, but while CWE is useful for many purposes, it does not provide a single, simple organizational structure, that is necessary for our purposes. "Top" lists, such as the "CWE/SANS top 25" and the "Open Web Application Security Project (OWASP) top 10," are helpful in identifying especially common weaknesses, but they make no attempt to cover all relevant objectives.

Instead, we have focused on identifying a set of detailed technical objectives that can help narrow the selection of appropriate tools and techniques.  We created this set of technical objectives by merging several accepted sources.  These sources included:

- The NSA Center for Assured Software (CAS), in particular their tools studies. This provided a foundational structure for breaking down weaknesses that were unintentional. [CAS 2012].

- National Vulnerability Database (NVD) CWE categories [NVD]. This supported the organizational structure by identifying non-overlapping areas that required coverage.

- Common Weakness Enumeration (CWE)/System Administration, Networking, and Security Institute (SANS) top 25. We used this to create more granular objectives, so that users could focus on particularly important technical objectives.

- PPP outline/Defense Acquisition Guidebook (DAG) Chapter 13 material.

- Open Web Application Security Project (OWASP).

- Web Application Security Consortium (WASC).

The associated matrix subdivides technical objectives, in some cases down to four levels, to further refine where different tool/technique types can be best applied.

Users may choose to extend the matrix. A program may have other technical objectives than the ones we list, for example, a project may wish to counter the risk of a supplier no longer supporting a product (e.g., a company has gone out of business or has retired a product). Alternatively, a project may choose to subdivide some technical objective(s) further, to help determine which tool/technique types cover which portions of a technical objective and where there are critical gaps.

Note that these technical objectives apply to mobile environments as well as other environments. For example, a lower tier in the objective "counter intentional-like/malicious logic" is embedded malicious logic (this is additional functionality not desired by the user, sometimes called a "Trojan horse"). This would be a relevant objective if a concern is a malicious mobile application having legitimate access to the microphone and network and acting as audio spyware invoked without permission or user knowledge. The technical objective category "other" includes countering "excessive power consumption," an issue that is rarely a key technical objective in other environments yet which can be a critical vulnerability if manipulated in mobile environments.

## B.  Technical Objectives – Main Categories

Table 4-1 lists the top-level technical objectives. Categories that are used directly or derived from [NSA 2012] are marked with an asterisk. The technical objectives are in many cases subdivided further. For example, the technical objective "counter unintentional-'like' weaknesses" is subdivided further into a second level.

**Table 4-1. Top-level Technical Objectives**

| | |
|---|---|
| 1. Provide design & code* quality | 1. Buffer Handling* |
| 2. Counter unintentional-like known vulnerabilities | 2. Injection* (SQL, command, etc.) |
| 3. Ensure authentication and access control* | 3. Encryption and Randomness* |
|     a. Authentication Issues | 4. File Handling* |
|     b. Credentials Management | 5. Information Leaks* |
|     c. Permissions, Privileges, and Access Control | 6. Number Handling* |
|     d. Least Privilege | 7. Control flow management* |
| 4. Counter unintentional-"like" weaknesses | 8. Initialization and Shutdown [of resources/ components]* |
| 5. Counter intentional-"like"/malicious logic* | |
|     a. Known malware | |
|     b. Not known malware | 9. Design Error |
| 6. Provide anti-tamper and ensure transparency | 10. System Element Isolation |
| 7. Counter development tool inserted weaknesses | |
| 8. Provide secure delivery | 11. Error Handling* & Fault isolation |
| 9. Provide secure configuration | |
| 10. Other | 12. Pointer and reference handling* |

\* indicates categories that are used directly or are derived from [NSA 2012].

Below is more information about the top-level technical objectives, including a brief list of some of their subdivisions. For brevity, many of the sub-categories do not include verbs, as their purpose can be inferred from the upper-level technical objectives they support. Categories that are used directly or derived from [NSA 2012] are marked with an asterisk:

1. Provide design and code quality*. Note that [NSA 2012] identifies "code quality" as a category; we have expanded this category to include design quality. Strictly speaking, most analysis tools and techniques can only provide information on the quality of the result; they cannot by themselves actually provide design or code quality. Nevertheless, the objective is to provide

information on or indicators of design and code quality, through the use of tools and techniques that attempt to measure the quality of candidate software.

2. Counter unintentional-like known vulnerabilities*. A component, including obsolete subcomponents, may have a known vulnerability. One approach for checking this is to examine the common vulnerability enumeration (CVE) list, which provides a standardized name convention. However, note that not all known vulnerabilities have CVEs. Vulnerabilities that appear to be unintentional, but are not known, are covered by other technical objectives (primarily the technical objective "counter unintentional-'like' weaknesses"). Vulnerabilities that appear to be intentional, are covered by the technical objective "counter intentional-'like'/malicious logic."

3. Ensure authentication and access control*. We have separated this category from countering unintentional-like and intentional weaknesses, because it is often unclear whether it is intentional or not. Also, it is important to subdivide this category, and we found it easier to discuss these subdivisions separately. The first three subcategories below were identified by [NVD]:

   a. Authentication Issues. This occurs "when an actor claims to have a given identity, [but] the software does not prove or insufficiently proves that the claim is correct" (this definition is from CWE-287).

   b. Credentials Management. This occurs when there is inadequate management of credentials (such as passwords or cryptographic keys).

   c. Permissions, Privileges, and Access Control. This includes granting resource access to another component that should not be allowed that access. Mobile environments often isolate applications from each other; applications that improperly and unintentionally allow other applications access to their services or resources are covered here. Note that the CWEs separate subjects (active processes) from objects (data). Thus applications that fail to authorize requests are missing authorization or may have improper/incorrect authorization. In contrast, applications that unwisely permit other applications to access resources (including critical resources possibly due to incorrect defaults) are considered to have permission issues.

   d. Least Privilege. Weaknesses in this category occur with improper enforcement of sandbox environments, or the improper handling, assignment, or management of privileges (this definition is from CWE-265).

4. Counter unintentional-like weaknesses. This technical objective covers weaknesses that are commonly unintentionally inserted by developers. However, we use the term "unintentional-like" because it is quite possible for a

malicious developer to intentionally insert a vulnerability that *appears* to be
unintentional. When examining a TOE, we can only guess at human intent.
Nevertheless, if an unintentional-like weakness is inserted intentionally, tools
and techniques for unintentional-like weaknesses can still be applied. We have
subdivided this further; for more detail (including definitions) on the
subdivisions marked with an asterisk, see [CAS]:

    a. Buffer Handling*,

    b. Injection* (SQL, command, etc.),

    c. Encryption and Randomness*,

    d. File Handling*,

    e. Information Leaks*,

    f. Number Handling*,

    g. Control Flow Management*,

    h. Initialization and Shutdown (of resources/components)*,

    i. Design Error (This covers design errors that lead to unintentional
vulnerabilities.),

    j. System Element Isolation (This covers errors involving allowing system
elements unfettered access to each other.),

    k. Error Handling* and Fault isolation,

    l. Pointer and Reference Handling*.

5. Counter intentional-like/malicious logic*. This objective covers weaknesses
that are commonly intentionally inserted (directly or indirectly), including
viruses, backdoors (logic that enables later unauthorized access), Trojan horses
(software that does something malicious in addition to its stated purpose,
including software that colludes to transmit data via covert channels), and so on.
As with unintentional-like weaknesses, we can only guess human intent. It is
possible in some cases for a vulnerability to appear malicious, yet not be
intentionally so. This objective is subdivided into:

    a. Known malware. This includes viruses with known signatures or patterns.

    b. Not known malware.

6. Provide anti-tamper and ensure transparency. The objective of anti-tamper is
"to impede unapproved technology transfer, alteration of system capability, or
countermeasure development" (per https://at.dod.mil/). This can be an
important goal for critical program information (CPI). This paper does not

specifically examine tools and techniques to implement this objective, since other organizations already focus on this. For more information on anti-tamper tools and techniques, see the DoD anti-tamper site at https://at.dod.mil/. The flip side of anti-tamper is transparency, in particular, the ability to easily examine in-depth a particular component. Non-transparent components (e.g., obfuscated ones) are more difficult to analyze, and thus it is more difficult to ensure that they have included good properties and avoided negative ones. It is possible to provide both, e.g., to require transparency from off-the-shelf suppliers for components to be used, then obfuscating custom components (or the combined fielded result) to impede technology transfer by adversaries.

7. Counter development tool-inserted weaknesses. Development and sustainment tools can themselves insert weaknesses, malicious or not. This objective covers countering this issue.

8. Provide secure delivery. This objective covers ensuring that software is delivered only to the intended recipient(s) with the requisite confidentiality, integrity, availability, and non-repudiation.

9. Provide secure configuration. This objective covers ensuring that the software, when installed and used, is securely configured for its context.

10. Other. This objective covers issues that do not easily fit into any other category. This includes excessive power consumption that can cause degradation of server performance and result in denial of service (loss of availability) in mobile applications.

# 5.  Types of Tools and Techniques

There is no widely accepted complete categorization of tools and techniques.  The NIST SAMATE project web page has a brief but limited list of tool categories.[5]  But this list is a work-in-progress; NIST has requested that a more organized taxonomy be developed.[6]  Another valuable source for categories of tools and techniques is [BAH 2009].

We have created a categorization of tools and techniques based on our own analysis, using sources such as our interviews and the NIST SAMATE project.  It is not the only possible categorization, and since it is incomplete, we do not call it a taxonomy.  Our goal is simply to create a *useful* set of categories that can be extended as required.  Unless otherwise noted, a category is a type of tool (it is primarily automated) and not a type of technique.   In general, we only included tool types where there is at least one commercially available tool; we granted some exceptions in the mobile space because that is a fast-paced environment.  For more about promising research efforts in this space, see [Wheeler 2012].  We expect that new types of tools and technologies could be added in the future to these categories, driven by innovation and commercialization (especially in the mobile environment).

This chapter presents our categorization of types of tools and techniques.  We have identified the following three major groupings of types of analysis tools and techniques:

- Static analysis: Examines the system/software without executing it, including examining source code, bytecode, and/or binaries.

- Dynamic analysis: Examines the system/software by executing it, giving it specific inputs, and examining results and/or outputs.

- Hybrid analysis: Tightly integrates static and dynamic analysis approaches.  For example, test coverage analyzers use dynamic analysis to run tests and then use static analysis to determine which parts of the software were not tested.  This grouping is used only if static and dynamic analyses are tightly integrated; a tool or technology type that is primarily static or primarily dynamic is put in those groupings instead.

---

[5]  http://samate.nist.gov/index.php/Tool_Survey.html

[6]  http://samate.nist.gov/index.php/Tool_Taxonomy.html

This grouping is similar to the groupings used by the 2015 Gartner Magic Quadrant for Application Security Testing (AST) report. That document groups tools into the following categories [Mello2015]:

- Static AST (SAST): "This technology analyzes an application's source and binary code for security vulnerabilities, typically at the programming or testing phases of the software lifecycle." This is essentially the same as our "static analysis" category.

- Dynamic AST (DAST): "This testing method analyzes applications while they're running. It simulates attacks against an application, analyzes the application's reactions to the attack, and then determines whether it's vulnerable or not." This is very similar to our "dynamic analysis" category, since we also emphasize execution. We do not strictly require attack simulation, though that is common.

- Interactive AST (IAST): "This technology combines elements of SAST and DAST simultaneously. It's typically implemented as an agent within the test runtime environment." We term this as "hybrid analysis."

- Mobile AST: "This method uses a combination of traditional SAST and DAST, and behavioral analysis using static and dynamic techniques to discover malicious or potentially risky actions the app may be taking unbeknownst to the user, which analyzes security vendors' static, dynamic, mobile, and interactive application testing capabilities." We do not create a separate category for these. Mobile is simply a particular kind of target, and can be addressed by static, dynamic, and/or hybrid analysis.

Most tools and techniques are subject to both false positives and false negatives. As a gross overgeneralization, static analysis tools have built-in a mechanism for reducing false negatives (missed vulnerabilities): they have access to the entire program's potential flow of data and control. However, static analysis tool developers have to contend with minimizing their tools' false positive rates, since a mistake in a particular area of code might not actually be exploitable in a wider context. Dynamic analysis tools have a built-in mechanism to help minimize false positives: a relevant data and control path must be executed before it will be detected. However, dynamic tool developers must contend with minimizing their tools' false negative rates (undetected vulnerabilities), because if a code path or data flow is not executed, the tools will typically be unable to report vulnerabilities along that path. Hybrid tools can inherit some of the strengths and weaknesses of static and dynamic approaches, depending on how they combine the approaches. These are generalizations; potential users must examine the particular tools they are interested in, and research continues to improve tools in all of these groupings. In general, it is better to use both static and dynamic approaches together (possibly

including hybrid approaches). We continue to expect that more hybrid analysis approaches will be developed in the future.

Some specific tools or tool suites combine multiple approaches. For example, some tool suites include both a static analysis tool (e.g., a source code weakness analyzer) and a dynamic analysis tool (e.g., a web application scanner). A tool may combine multiple dynamic approaches, or multiple static approaches, or multiple hybrid approaches. A tool may even combine multiple static, multiple dynamic, and/or hybrid approaches. We expect that there will be an increase in the number of tools and/or tool suites that use multiple analysis approaches. Some tool suites designed to support analysis of mobile applications use a large number of different approaches, and are difficult to fully categorize; this may reflect the fact that mobile environments are newer and have tools which are rapidly evolving. We also expect this trend to continue.

For each of these major groups (static, dynamic, and hybrid), we have identified a number of types of tools and techniques. The following subsections briefly identify and describe these types. In a few cases we show groupings of types, and then the types themselves as sub-lists. More detail about each type can be found in the fact sheets in Appendix C. The fact sheets include a list of examples of tools that include the given tool/technique. As previously stated, the lists are illustrative and not all-inclusive and no endorsement of any particular tool is implied. Also note that not all of the specified tools in a category necessarily address a technical objective as shown in the SOAR Matrix, but at least one tool in that category does. Users should evaluate specific tools as appropriate once they know what technical objectives they wish to address.

In some cases a tool/technique type can be used in different ways that substantially affect its effectiveness for some technical objectives. In these cases we discuss the general tool/technique type once and show these different ways as separate columns in the matrix. We count the number of matrix columns when we report the number of tool/technique types, since when making decisions it is comparing these different columns that matters.

Different names are used in industry for the same type of tool or technique, and in some cases the same name is used for different types. For our purposes we selected one name as the primary name for a given tool/technique type. We preferred names that were short, descriptive, and used in practice. Note that the tool/technique type names do not necessarily have the same grammatical part of speech, since in practice people refer to these tools and techniques in different ways. That said, we often used nouns when referring to a tool, and other forms of speech when referring to techniques or groups of tools. The fact sheets identify some of the alternative names in use.

In the past, addressing assurance has often focused on source code weakness analysis tools (a.k.a. source code security analyzers, static analysis code scanners, static

application security testing (SAST) tools, or code weakness analysis tools).  These tools can be very useful.  However, as should be clear from the long list of tool/technique types in this paper, other types of tools/techniques can be applied to address assurance.

## A.  Static Analysis

The following are static analysis tool/technology types:

1.  Attack modeling. Attack modeling analyzes the system architecture from an attacker's point of view to find weaknesses or vulnerabilities that should be countered.

2.  Source code analyzers[7] is a group of the following tool types:

    a.  Warning flags. Warning flags are mechanisms built into programming language implementations and platforms that warn of dangerous circumstances while processing source code.

    b.  Source code quality analyzer.  Source code quality analyzers examine software source code and search for the implementation of poor coding or certain poor architecture practices, using pattern matches against good coding practices or mistakes that can lead to poor functionality, poor performance, costly maintenance, or security weaknesses depending on context.  There is now a preponderance of evidence that higher-quality software (in general) tends to produce more secure software [Woody 2014].  These kinds of tools are often less expensive than some other kinds, and can often be applied earlier in development, providing good reasons to use them even when the focus is to develop secure software.

    c.  Source code weakness analyzer. Source code weakness analyzers examine software source code and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses).  This kind of tool is also called a "source code security analyzer," "static application security testing" (SAST) tool, "static analysis code scanner," or "code weakness analysis tool."  We've chosen the name "source code weakness analyzer" because this name more clearly defines what this type of tool does and distinguishes it from other types of analysis.

    d.  Context-configured source code weakness analyzer.  This configures a source code weakness analyzer specifically for the product being evaluated (e.g., by adding many additional rules).

---

[7] For the purposes of this paper, "source code analyzer" is a *group* of tool types; the lettered items below are the tool/technique types.  A person who performs manual review of source code could also be considered a "source code analyzer," but for our purposes we group manual review processes separately.

e. Source code knowledge extractor for architectural, design, and mission layer information. This extracts information such as the architecture and design from the source code to aid analysis. Note that knowledge extractors can be used in many other ways; in particular, a knowledge extractor can be used as the technical baseline for implementing source code quality analyzer or a source code weakness analyzer. In those cases extractors fall into other categories; this category focuses solely on using extractors to obtain architectural, design, and mission layer information.

f. Requirements-configured source code knowledge extractor. This configures a source code knowledge extractor to analyze a particular system.

3. Binary/bytecode analysis is a group of the following tool types:

a. Traditional virus/spyware scanner. Traditional virus/spyware scanners search for known malicious patterns in the binary or bytecode. Note that modern anti-virus programs also perform behavioral analysis; this capability is (for our purposes) rolled into intrusion detection systems (IDS)/intrusion prevention systems (IPS).

b. Quality analyzer. Binary/bytecode quality analyzers examine the binary or bytecode (respectively) and search for the implementation of poor coding or certain poor architecture practices, using pattern matches against good coding practices or mistakes that can lead to poor functionality, performance, costly maintenance, or security weaknesses depending on context. Note that this is similar to source code quality analyzers, except the analysis is performed on a binary or bytecode. There is now a preponderance of evidence that higher-quality software (in general) tends to produce more secure software [Woody 2014].

c. Bytecode weakness analyzer. Bytecode weakness analyzers examine binaries and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). Note that these are similar to source code weakness analyzers, except the analysis is performed on bytecode.

d. Binary weakness analyzer. Binary weakness analyzers examine binaries and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). Note that these are similar to source code weakness analyzers, except the analysis is performed on a binary.

e. Inter-application flow analyzer. These tools examine the control and/or data flows of a set of applications, identifying their communication interfaces

(such as Android intents [Android intents]) and permissions, and then identify flows that violate the security policy.

f.   Binary/bytecode simple extractor. Binary/bytecode simple extractors are simple tools that report simple facts about binary executables or bytecode, or perform trivial analysis of them. e.g., they may report the text strings within a binary or bytecode.

g.   Compare binary/bytecode to application permission manifest.  Examine the binary/bytecode to determine what permissions the application attempts to use, and compare that to the permissions actually requested in the application permission manifest.  Note that permissions in this context are the privileges granted to an applications, not the permissions set on objects such as files or memory.

4.   Obfuscated code detection.  Obfuscated code detectors detect when code is rendered obscure.  They may be applied to source code (e.g., JavaScript), bytecode, or executables.  Obfuscation may be used to counter reverse-engineering of critical or proprietary technology, but it can also be used to counter analysis by other assurance tools.  Thus, obfuscated code may represent an increased risk of unintentionally vulnerable or intentionally malicious code.

5.   Binary/bytecode disassembler. Binary/bytecode disassemblers recover higher-level constructs from lower-level binaries and bytecode, which can then be analyzed by people or automated tools.

6.   Human review.  This is typically done with source code, but it can also be done with binary or bytecode (often this is generated by a binary or bytecode disassembler, as noted above).  Note that human reviews can apply to products other than code, including requirements, architecture, design, and test artifacts.  Human reviews include the following more specific types of techniques:

a.   Focused manual spot check.  This specialized technique focuses on manual analysis of code (typically less than 100 lines of code) to answer specific questions.  For example, does the software require authorization when it should? Do the software interfaces contain input checking and validation?

b.   Manual code review (other than inspections).  This specialized technique is the manual examination of code, e.g., to look for malicious code.

c.   Inspections (Institute of Electrical and Electronics Engineers (IEEE) standard).  IEEE 1028 inspection is a systematic peer examination to detect and identify software product anomalies.

d.   Generated code inspection. This technique examines generated binary or bytecode to determine that it accurately represents the source code.  For

example, if a compiler or later process inserts malicious code, this technique might detect it. This is usually a spot check and not performed across all of the code.

7. Secure platform selection is a group of the following tool types:

   a. Safer languages. This is selecting languages, or language subsets, that eliminate or make it more difficult to inadvertently insert vulnerabilities. This includes selecting memory-safe and type-safe languages.

   b. Secure library selection. Secure libraries provide mechanisms designed to simplify developing secure applications. They may be standalone or be built into larger libraries and platforms.

   c. Secured operating system (OS). A secured OS is an underlying operating system and platform that is hardened to reduce the number, exploitability, and impact of vulnerabilities.

8. Origin analyzer. Origin analyzers are tools that analyze source code, bytecode, or binary code to determine their origins (e.g., pedigree and version). From this information, some estimate of riskiness may be determined, including the potential identification of obsolete/vulnerable libraries and reused code.

9. Digital signature verification. Digital signature verification ensures that software is verified as being from the authorized source (and has not been tampered with since its development). This typically involves checking cryptographic signatures.

10. Configuration checker. Configuration checkers assess the configuration of software to ensure that it meets requirements, including security requirements. A configuration is the set of settings that determine how the software is accessed, is protected, and operates.

11. Permission manifest analyzer. Permission manifest analyzers are tools that analyze the application's permission manifest and estimate level of risk (possibly using policy requirements to determine what is more or less risky). This requires that there be a permission manifest (e.g., like Android's), and is similar to a configuration checker. Note that this manifest analysis is done without reference to the code itself.

12. Development/sustainment version control. Version control tools record and track who made which change, and when the change was made. This information can ease identification of who may have inserted vulnerabilities (unintentional or malicious). Version control creates a deterrent for inserting vulnerabilities and a starting point for remediation.

13. Obfuscator.  An obfuscator tool takes source, bytecode, or binary and transforms it into something difficult to understand or reverse-engineer.

14. Rebuild and compare.  The rebuild and compare technique rebuilds a bytecode or binary from its purported source code, and then determines whether the rebuilt version is equivalent to the bytecode or binary provided.  If it is, then the bytecode or binary corresponds to its purported source code (given certain assumptions).

15. Formal methods/correct-by-construction. Formal methods are the use of mathematically rigorous techniques and tools for the specification, development, and verification of software and hardware systems [Butler].  We provide more information in Appendix C, but as explained in the appendix, they are not listed in the matrix or in the count of tool/technique types.

## B.  Dynamic Analysis

The following are dynamic analysis tool/technique types (this work assumes that traditional functional testing is already being performed separately, e.g., functional qualification testing, and those related tools/techniques are excluded from this research, including traditional functional testing of authentication and authorization mechanisms to ensure that authorized users can access the component):

1. Network scanner.  A network scanner identifies network components (nodes) and network connections (ports) by actively interacting with other network components on the network.  Using a network scanner is often a first step in using other tools, such as network vulnerability scanners and IDSs, and they are often packaged together.

2. Network sniffer.  A network sniffer observes and records network traffic.  This information can then be analyzed to identify unexpected network traffic, perform trend analysis, and so on.

3. Network vulnerability scanner. A network vulnerability scanner sends network traffic to a network node, or a service on a network node, to determine whether it meets security policies and to identify any known vulnerabilities.

4. Host-based vulnerability scanner. A host-based vulnerability scanner examines a host system configuration for flaws and ensures that the host configuration meets certain predefined criteria.  It may also verify that the audit mechanisms work.  This type of tool can be used both before deployment and during operations.

5. Host application interface scanner.  A host application interface scanner identifies the various host-based interfaces of applications.

6. Application-type-specific vulnerability scanner.  An application-type-specific vulnerability scanner sends data to an application, to identify both known and new vulnerabilities.  It may look for known vulnerability patterns (a.k.a. weaknesses) and anomalies.  This is a group of the following tool types:

    a. Web application vulnerability scanner.  A web application vulnerability scanner automatically scans web applications for potential vulnerabilities.  They typically simulate a web browser user, by trawling through URLs and trying to attack the web application.  For example, they may perform checks for field manipulation and cookie poisoning [SAMATE].

    b. Web services scanner.  A web services scanner automatically scans a web service (as opposed to a web application), e.g., for potential vulnerabilities. [SAMATE]

    c. Database scanner.  Database scanners are specialized tools used specifically to identify vulnerabilities in database applications. [SAMATE]  For example, they may detect unauthorized altered data (including modification of tables) and excessive privileges.

7. Fuzz tester.  A fuzz tester provides invalid, unexpected, or random data to software, to determine whether problems occur (e.g., crashes or failed built-in assertions).  Note that many scanners (listed above) use fuzz testing approaches.

8. Framework-based fuzzer.  A framework-based fuzzer creates inputs and observes results, as with traditional fuzzing, but instruments the underlying framework to help identify and select what inputs would be most relevant to test.

9. Negative testing.  For the purpose of this paper, negative testing is a technique that includes, in the regression test suite, many tests that should fail if the security mechanisms work properly.  This is not a tool, but a test-case-generation criterion for existing test tools.  A simple example of negative testing is a test that tries to use a seven-character password for a system that requires at least an eight-character password.

10. Digital forensics.  Digital forensics tools are tools that support "the use of … methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence … for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations" [Palmer 2001].

11. Intrusion Detection System (IDS)/Intrusion Prevention System (IPS).  An IDS monitors network or system activities for malicious activities or policy

violations and reports them. An IPS also monitors, but instead of just reporting activities or violations, it actively prevents or remediates them. This paper considers IDS/IPS a single tool type. Note, however, that an IDS/IPS can be implemented in one of two ways (a tool can combine both of these approaches in a single product):

a. Network-based IDS/IPS. A network-based IDS or IPS monitors network traffic to perform its monitoring, prevention, and/or remediation for malicious activities or policy violations.

b. Host-based IDS/IPS/Integrity checker. A host-based IDS, IPS, or integrity checker monitors data other than network traffic (such as files, registry values, and program input/output) for malicious activities or policy violations.

12. Automated detonation chamber (limited time) automatically isolates a program (including running multiple copies in virtual machines), executes it, detects potentially malicious or unintentionally vulnerable activities, and then reports its findings prior to the software's deployment. In contrast, we use the broader term "monitored execution" to refer to broader processes that use many tools/techniques (including manual techniques) to isolate software and detect malicious activities. It is often useful to run software in isolation (to limit damage), but in this case the software is run for some limited time to analyze the software's behavior. Previous versions of this paper called this "automated monitored execution (limited time)."

13. Forced path execution. Forced path execution runs a program and forces execution of all (control flow) paths, even if the test inputs would not normally cause it to do so, and monitors what happens to detect possible undesired behavior.

14. Firewall. A firewall limits network access based on a set of rules. A firewall can be network-based (e.g., used as a gateway into a network) or host-based (e.g., limit access between one host and a network). They typically check traffic against signatures and anomalies. This paper considers firewall a single tool type, but there are at least two variants of firewall:

a. Network firewall. This limits access at the network level.

b. Web application firewall. A web application firewall examines network traffic at the web application level to detect and/or limit damage. Its deeper inspection than that of typical network firewalls or IPSs can protect web applications/servers from web-based attacks that IPSs cannot prevent.

15. Man-in-the-middle attack tool.  This type of tool attempts to intercept and perform a man-in-the-middle attack on the application.  This can be at the network level, or in lower-level application communication protocols.

16. Debugger.  A debugger is a tool that enables observation and control of a program under execution.  This can include the ability to execute the program step by step, and to observe internal states and results.

17. Fault injection. These techniques insert faults into software to enable better testing.  This is a group of the following tool types:

    a.  Source code fault injection.  "Source code fault injection tools provide a mechanism through which source code can be instrumented to induce the code to follow control paths that would be otherwise difficult to test for." [BAH 2009]

    b.  Binary fault injection.  "Binary fault injection tools provide mechanisms through which safety- or security-related faults can be sent to the application while it is running… Unlike source code fault injection, binary fault injection does not require knowledge of the application's source [code]." [BAH 2009]

18. Logging systems. A logging system records events, and their times, to provide an audit trail that can be used to understand software activity and diagnose problems.  The "syslog" service is an example. This information may be sent to a Security Information and Event Management (SIEM) system.

19. Security Information and Event Management (SIEM).  "SIEM technology provides real-time analysis of security alerts generated by network hardware and applications."  [Dr. Dobbs 2007]

## C.  Hybrid Analysis

The following are hybrid analysis tool/technology types:

1. Test coverage analyzer.  Test coverage analyzers are tools that measure the degree to which a program has been tested (e.g., by a regression test suite).  Common measures of test coverage include statement coverage (the percentage of program statements executed by at least one test) and branch coverage (the percentage program branch alternatives executed by at least one test).  Areas that have not been tested can then be examined, e.g., to determine whether more tests should be created or whether that code is unwanted.

2. Hardening tools/scripts. This type of tool modifies software configuration to counter or mitigate attacks, or to comply with policy.  In the process, it may detect weaknesses or vulnerabilities in the software being configured.

3. Execute and compare with application manifest. Run an application with a variety of inputs to determine the permissions it tries to use, and compare that with the application permission manifest.

4. Track sensitive data. Statically identify data that should not be transmitted or shared (e.g., due to privacy concerns or confidentiality requirements), then dynamically execute the application, tracking that data as tainted to detect exfiltration attempts.

5. Coverage-guided fuzz tester. Use code coverage information to determine new inputs to test.

6. Probe-based Attack with Tracked Flow. Observe normal behavior while tracking data and control flows within the program (possibly through several tiers), send probing inputs to determine patterns of behavior that might indicate a potential vulnerability, then based on these patterns, perform simulated attacks to identify actual vulnerabilities.

7. Track Data and Control Flow. Track data and control flows from inputs and other data sources to data sinks, and report when rules (predefined or user defined) are triggered indicating a potential vulnerability.

## D. Advantages of Combining Tools and Techniques

As shown in Appendix E, the Software SOAR Matrix, no one type of tool or technique can address all possible technical objectives. Some tool/technique types only address one or a few specific technical objectives, but are highly effective for that scope. Those that have broader applicability may have challenges (e.g., some can be more costly or require deeper expertise). Thankfully, static, dynamic, and hybrid analysis tools and techniques can be combined to alleviate some of these limitations.

Automated tools and manual techniques are often interleaved to achieve a higher-quality evaluation. For example, a human may analyze the results of a source code weakness analyzer and modify the tool configuration to reduce the number of false positives. Similarly, when applying a framework-based fuzzer, the tool may automatically generate many inputs for testing, yet a human may intervene and provide specific inputs to guide the testing (to increase test coverage).

It is often useful to combine multiple types of tools and techniques, and in many cases it is useful to combine multiple tools of the same tool type. Almost all tools have many false negatives (missed reports). For example, [CAS20111] examined many source code weakness analyzers and found that any one tool tends to find a minority of the vulnerabilities of an application, even for just the types of vulnerabilities the tool is designed to find.

**Figure 5-1. Conceptual illustration of using multiple tools and techniques**

Figure 5-2 is a conceptual illustration of the advantages of using multiple tools and techniques, particularly when they use different approaches. The arrows represent potential risks, including exposed vulnerabilities in the software, and the screens represent tools and techniques applied by a project. No one tool or technique addresses all technical objectives, and almost all only find a fraction of the vulnerabilities and other issues they address. Thus, applying multiple tools and techniques is more helpful. Each tool or technique contributes to meeting technical objectives (and thus reducing overall risk).

Note that even when applying multiple tools and techniques there is no guarantee that all technical objectives (e.g., vulnerability removal) will be perfectly achieved, so there is a still need to monitor operational systems, counter active attacks, and respond/recover. However, if software is extremely vulnerable, such monitoring, countering, and response/recovery is difficult to achieve, so even imperfect removal of vulnerabilities is worthwhile.

## E.  Processes to Combine Tools and Techniques

There are various ways to combine tools and techniques. Processes to combine different types of analysis include monitored execution (aka "detonation chamber"), SwA correlation, penetration testing (aka "pen testing"), audit processes, problem/bug/incident report analysis, and assurance case development. For our purposes, these are not considered tools or techniques. Instead, these are larger processes that may use many of these tools and techniques. These larger processes may also meet multiple technical objectives, as discussed below.

### 1. Monitored Execution

*Monitored execution* (aka using a "detonation chamber") is a process that runs software (the target of evaluation (TOE)) in an isolated system to detect suspicious activity. This paper discusses, as a specific approach, using an automated detonation chamber for a limited time. However, there are other ways to perform monitored execution. Monitored execution can combine many different tools and (manual) techniques, and it can be applied continuously or for an extended period of time. Note that monitored execution may be conducted during development, sustainment, or operations. In all cases, users of the monitored execution process isolate the software and then attempt to detect problems while executing it, so when applying this approach consider:

1. *Isolation*. Users of this process may use various approaches to isolate the TOE. They may choose software-based isolation approaches such as sandboxes, wrappers, debuggers, or virtual machines. Alternatively, users may install the TOE on a "real" but isolated system and later uninstall the software (often by restoring the system to a "known good" state). The latter approach is called a "sacrificial installation" and can be useful if there is concern that the TOE contains malicious software that detects isolation mechanisms (e.g., counter-debugger or counter-virtual machine mechanisms) and behaves differently under them. A challenge for sacrificial installations is ensuring that all malicious software has truly been erased (e.g., from computer and peripheral firmware).

2. *Detection*. Users of this process may use various tools and techniques to detect problems (including unexpected changes, unexpected behavior, or unexpected results). Examples of network-based tools include network-based IDSs and network sniffers. Examples of host-based tools include integrity checks of files, registry entries, disk boot blocks, host-based IDSs, and so on.

Again, note that we use the term "automated detonation chamber (limited time)" to refer to a specific tool type that automatically isolates a program and/or data (including running multiple copies in virtual machines), executes/processes it, detects potentially malicious or unintentionally vulnerable activities, and then reports its findings (typically prior to the software's deployment). In contrast, we use the broader term "monitored execution" where a variety of tools and/or techniques are combined to perform isolation, detection, and analysis. A monitored execution process may use automated monitored execution (limited time) as part of its larger process.

### 2. SwA Correlation

*SwA correlation* is the process of correlating the results of multiple SwA tools and techniques. This can be done manually or through a SwA correlation tool; SwA

correlation tools have the advantage of being much faster at larger scale. Other terms for SwA correlation tools include "application vulnerability management tool" and "software vulnerability assessment tool." Examples of SwA correlation tools include CodeDX (from Secure Decisions), ThreadFix (from the Denim Group), SonarQube (from SonarSource), and TOIF (from KDM Analytics). An ideal SwA correlation tool would support many types of tools and techniques (e.g., static, dynamic, and hybrid), a large number of common tools, and perform the following functions:

- Aggregation – Collects and displays all results from automatic tool scans and manual techniques

- Normalization – Interprets the semantics from each tool/technique and maps them to a normalized flaw type or CWE

- De-duplication – Groups the same weaknesses reported by multiple tools/techniques into one finding

- Prioritization – Automatically assigns a severity level to individual and grouped findings

- Weakness Location Display – Provides a full context display of the discovered weakness in the context of the rest of the code

## 3. Penetration Testing

*Penetration testing* is "A test methodology in which assessors, typically working under specific constraints, attempt to circumvent or defeat the security features of an information system" [CNSS 4009]. In short, penetration testing performs a simulated attack.

## 4. Audit Processes

*Audit processes* are the "independent review and examination of records and activities to assess the adequacy of system controls and ensure compliance with established policies and operational procedure" [CNSS 4009]. Audit processes can leverage firewalls, IPSs, and logging systems to extract relevant information for assurance analysis. The information can be synthesized and analyzed to identify characteristics that tie into technical objectives.

## 5. Problem/bug/incident report analysis

*Problem/bug/incident report analysis* is examining the problem reports, bug reports, incident reports, and related information to identify overall problems and trends.

**6.  Assurance Case**

An *assurance case* is "a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's [security] properties are adequately justified for a given application in a given environment" [IATAC 2007].  By itself, an assurance case is not a tool or technique in the sense we have defined these terms.  Instead, an assurance case is a way of organizing evidence (some of which may be extracted using tools and techniques), through various arguments, to justify a set of claims.

## F.  Excluded Tools and Techniques

We have excluded these categories of tools and techniques as being out of scope for this paper:

1.  General-purpose software test tools and test frameworks.  Having an automated test framework is extremely important for software reliability.  A good automated test suite is also an important aid for security, because it enables projects to quickly update vulnerable components.  Modern systems typically include a vast amount of reused software.  If a vulnerability is found in a component (say using an origin analysis tool), a good automated test suite can enable rapid updating of the component and redistribution of the updated system.  However, while this category of tool is related to the focus of this paper, they are different enough that we have excluded them.  We do include some related tools or techniques:

    − Test coverage measurement tools. These help measure the quality of the test suite.

    − "Negative" testing, that is, tests to ensure that the system does not do what it is not supposed to do (e.g., that the system rejects invalid security certificates).  Strictly speaking, a good automated test suite would simply include such tests.  However, many test suite developers fail to include these kinds of important security tests, so we specifically list them as a tool/technique.

2.  Combinatorial testing and other related mechanisms for selecting a (relatively) minimal set of test cases that meet certain criteria.  Examples include Automated Combinatorial Testing for Software (ACTS)[8] and covering arrays.[9]

---

[8] http://csrc.nist.gov/groups/SNS/acts/index.html

[9] http://www.jmp.com/support/help/Covering_Arrays.shtml

3. Threat intelligence. Gartner defines threat intelligence as "evidence-based knowledge, including context, mechanisms, indicators, implications and actionable advice, about an existing or emerging menace or hazard to assets that can be used to inform decisions regarding the subject's response to that menace or hazard." [Lee2014] This information can indirectly guide how to efficiently identify vulnerabilities, or it may suggest the technical objectives to select. However, since this is not particularly focused on analyzing specific software, we have excluded this area from this paper.

Although these may be very useful to some programs, we have excluded them from this paper because they are out of its scope.

# 6. Software Component Context

A TOE often consists of different types of software components with different characteristics that require distinct handling. Thus, the set of technical objectives, and/or the applicable tools and techniques, may be different as well. The following subsections identify some general factors that shape the context, and then briefly list the software component contexts provided in the PPP outline template that may significantly affect the context. Other factors may be relevant as well.

## A. General Factors

Factors that shape the context include:

- Mission criticality. Is the component critical per a criticality analysis? This decision must be based on the mission and environment.

- Critical program information (CPI). Is the component, or some of its technology, considered CPI?

- Amount of custom development. Is the component considered custom (developmental), off-the-shelf (OTS), or a mixture? OTS can include government off-the-shelf (GOTS) or commercial off-the-shelf (COTS). The term COTS includes nearly all proprietary software and open source software.

- Information availability. What information is available on the software; in particular, is source code available? Is enough information available that the software could be modified and rebuilt? This is important since several tools/technique types require source code, or even the ability to make changes. More information is often available for custom software, but this is not always true.

- Technologies used. What technologies are being used to implement the component, for example, what programming languages and platforms are being used? For example, binary analysis tools are irrelevant for programming languages implemented by interpreters. Many tools work on only specific programming languages. Many other tools can only be applied to specific types of applications, such as web applications, mobile applications, or embedded applications. Some tools can be applied to only specific platforms, such as tools that can be applied to only Windows, Linux, or Android mobile applications.

- Supply chain exposure (per threat analysis). Is the supplier perceived as risky, is there enough visibility into the supplier and their supply chain to determine risk. How well is the supplier protected from external malicious influences?

- Operational or developmental usage. Will the software be operationally deployed, or will it be used in-house for development, test, etc.?

## B. PPP Contexts

"Program Protection Plan Outline & Guidance" [DASD(SE) 2011], section 5.3.3 identifies the following software component contexts, where "developmental" means "custom":

- Developmental and Operational:
    - Developmental CPI,
    - Developmental Critical Function,
    - Other Developmental,
    - COTS CPI and Critical Function,
    - COTS (other than CPI and Critical Function) and non-developmental items (NDI).

- Development environment:
    - (C) Compiler,
    - Runtime libraries,
    - Automated test system,
    - Configuration management system,
    - Database.

More information is available in [DASD(SE) 2014].

# 7.    Program Protection Plan Roll-up

The plans for vulnerability analysis, including information on the planned tools and techniques for analyzing software, can be rolled up (summarized) as part of a PPP as described in [DASD(SE) 2011].  In particular, the PPP table "Application of Software Assurance Countermeasures" can be viewed as a roll-up summary.

As noted in [DASD(SE) 2011], "Program Protection is the integrating process for managing risks to advanced technology and mission-critical system functionality from foreign collection, design vulnerability or supply chain exploit/insertion, and battlefield loss throughout the acquisition lifecycle.  The purpose of the PPP is to help programs ensure that they adequately protect their technology, components, and information…. The process of preparing a PPP is intended to help program offices consciously think through what needs to be protected and to develop a plan to provide that protection."

The PPP includes, and is affected by the results of, the criticality analysis and threat analysis; these help define the software context and focus the analysis. The results of various analysis tools and techniques provide inputs to vulnerability analysis.   The vulnerability analysis also includes information about processes and people (individuals and organizations) as appropriate.  These are all rolled up into the overall PPP structure.

Note that the PPP "Application of Software Assurance Countermeasures" divides software into three major PPP categories:

- Development process. This covers custom software developed for use in an operational setting.

- Operational system. This covers off-the-shelf (OTS) software developed for use in an operational setting.

- Development environment. This covers developed software not intended for deployment in an operational setting, e.g., the development, sustainment, or test environments.

Since different programs may choose to select different tools and techniques, the list of tools and techniques that should be rolled up will vary.  However, nearly all of the PPP template categories can be filled in with information based on these tools and techniques.

Table 7-1 lists the PPP categories in [DASD(SE) 2011], identifies the type of PPP category and then identifies which tool/technique types can be used to provide this information (where appropriate).  PPP categories may be tool roll-ups (which summarize

information from certain kinds of tools), objective roll-ups (which summarize information from certain technical objectives), or information roll-ups (which provide information about the software under evaluation, e.g., if there is source code available).

**Table 7-1. PPP Category Roll-ups**

| PPP Category | PPP category information roll-up type | Static | Dynamic | Hybrid |
|---|---|---|---|---|
| Static Analysis | Tool/technique | All static analysis tools, e.g., Warning flags, Source code quality analyzer, Source code weakness analyzer (SCWA), context-configured SCWA, … | - | - |
| Design Inspect | Tool/technique | Human review, knowledge extractors | | |
| Code Inspect | Tool/technique | Human review, Warning flags, Source code quality analyzer, Source code weakness analyzer (SCWA), context-configured SCWA, … | | |
| CVE | Objective | All already-known vulnerabilities | All known vulnerabilities | All known vulnerabilities |
| CAPEC | Tool/technique | Attack modeling | | |
| CWE | Objective | All | All | All |
| Pen Test | Tool/technique | Selective use | Comprehensive | Selective use |
| Test Coverage | Tool/technique | - | Fuzz testing, Application-specific vulnerability scanners | Test coverage analyzers |
| Failover Multiple Supplier Redundancy | Tool/technique | Human review | General testing (for interop/ replace), negative testing | |
| Fault Isolation | Tool/technique | Component isolation | Fault injection | |
| Least Privilege | Objective | Focused manual spot check, configuration checkers, knowledge extractors | IDS, Functional test | Access control rules and enforcement |
| System Element Isolation | Objective | Focused manual spot check, configuration checkers, knowledge extractors | IDS, Functional test | Access control rules and enforcement |

| PPP Category | PPP category information roll-up type | Static | Dynamic | Hybrid |
|---|---|---|---|---|
| Input checking/ validation | Objective | Warning flags, Source code quality analyzer, Source code weakness analyzer (SCWA), context-configured SCWA, … | Application-type-specific vulnerability scanners, fuzz testing, negative testing | Test coverage analyzers, host-based system scanner |
| SW load key | Objective (combining Software delivery integrity and Anti-tamper) | Digital signature check | | Anti-tamper |
| [Development tool] Source | Information | (Helps determine what analysis tools can be used) | | (Helps determine what tools can be used) |
| [Development tool] release testing | Objective | | | |
| Generated code inspection | Objective | Human review - Generated code inspection; binary/bytecode simple extractor; Source code weakness analyzer; binary/bytecode weakness analyzer; knowledge extractor | Host-based IDS; Web application vulnerability scanner | Test coverage analyzers |

Note that DAG chapter 13 [DAG] explains "Failover Multiple Supplier Redundancy" by stating that "Identical code for a failed function will most likely suffer the same failure as the original. For redundancy in software, therefore, a completely separate implementation of the function is needed. This independence reduces the probability that the failover code will be susceptible to the same problem." Although this approach can be effective in hardware, experiments have shown that multiple in-line components at run-time do not provide the expected level of reliability in software [Knight 1986]. This approach does not necessarily, however, require multiple in-line components at run-time. It can be implemented by applying open systems approaches, that is, using open standards as interfaces so that a module can be replaced if necessary in some future release (e.g., because it is a malfunctioning, malicious, excessively expensive, etc.). If the program uses an open systems approach, it will need to perform testing with multiple implementations to ensure that multiple suppliers can be used in the future.

# 8.    Application

This section provides recommendations on how to apply the process recommended in this document.  In particular, this section provides tips on selecting technical objectives and selecting combinations of types of tools and techniques.

## A.    Selecting Technical Objectives

To select technical objectives, first consider the missions that the system/component supports and the role it plays.  In particular, what is the impact of failure or subversion on the mission(s)?   If the information the system or component processes loses its confidentiality or integrity, what is the impact?  Who might attack the system, and with what level of resources?  The goal is to estimate the likelihood of attack, the probability of success, and the resulting impact if there is no change to the development process and then to select changes to manage those risks.

First, decompose the system or component into smaller components until their differences are not distinct for analysis.  This can help focus effort on the parts that matter most.  Then, identify critical components (which may merit additional analysis).

For each component, consider the following:

1.  Consider what kind of component it is.

    a.  Is it a server-side web application, embedded, or something else?  If it is a server-side web application or embedded, consider adding all rows selected under the appropriate "filter for context" column.

    b.  Is it a critical component (as determined by a criticality analysis)?  What will be the impact if it fails or is subverted?  If the impact is high, then typically there will be more technical objectives and more tools and techniques to address them, and it will have a higher priority.

2.  Identify the most common kinds of vulnerabilities that apply to this software, and add countering them to the list of technical objectives.  Examples of these common vulnerabilities include buffer overflows and SQL injections.  If the software already exists and extensive data about its previous vulnerabilities has been collected, use that data (this data would typically be collected by combining past analyses and operational reports).  Otherwise, use lists of common kinds of vulnerabilities for that kind of software and platform.  For

web applications, a widely used list is the Open Web Application Security Project (OWASP) top 10. Otherwise, a common list to use is the SANS/CWE top 25.

3. Examine the technical objective categories (listed below), to determine which (other) areas matter for that system's purposes. Examples of these categories include "provide design and code quality" and "counter unintentional-like known vulnerabilities." Note that many of the common kinds of vulnerabilities to be countered, identified in the previous step, will already be identified as part of the technical objective category "counter unintentional-'like' weaknesses."

4. Prioritize. Where necessary, reduce the objectives. This should be in consultation with all stakeholders, including the authorizing office (AO) (formerly called the Designated Approving Authority (DAA)).

When examining the 10 topmost technical objective categories (as described above), consider the following:

1. Provide design and code quality: Most systems will want to include this as a technical objective. Low quality tends to lower security and make maintenance more expensive. In many cases, where source code is available this would prompt selection of a source code quality analyzer.

2. Counter known unintentional-like vulnerabilities: Systems that incorporate third-party components (which today is nearly all systems) should include this technical objective. Indeed, most systems today are predominantly implemented using third-party components, so in most systems this would be an important technical objective.

3. Ensure authentication and access control: Systems that implement authentication and access control should include these as appropriate. Least privilege, in particular, can reduce the impact caused by an attacker who finds a vulnerability.

4. Counter unintentional-"like" weaknesses. In particular:

   a. If you are using C, C++, or assembly programming languages, you should include the technical objectives for "buffer handling." Buffer handling errors often lead to vulnerabilities, and these languages do not provide automatic protection against them. Note that choosing "safer languages" can essentially eliminate these problems.

   b. If you are using a database, include the technical objective for countering "SQL injection."

c. The "design error" objective would normally be included by any system as a technical objective.

5. Counter intentional-"like"/malicious logic. Determine the likelihood that a custom or third-party component might have embedded malicious logic and its impact, and manage where necessary. For custom development, many organizations limit themselves to cleared personnel, as a risk-reduction mechanism. Note that few tools and techniques address unknown malicious logic.

6. Provide anti-tamper and ensure transparency. If your program requires it, identify anti-tamper as a technical objective. Transparency is the ability to easily examine in depth a particular component; transparency of third-party components can be valuable for supporting risk-reduction measures, but requiring it can reduce the number of available components (since some suppliers will be unwilling to do this).

7. Counter development-tool-inserted weaknesses. Development tools can themselves insert weaknesses. These are important to consider, but only if more easily accessed attacks are addressed.

8. Provide secure delivery. This should normally be included. This would typically prompt selection of digital signature verification.

9. Provide secure configuration. This should normally be included.

10. Other. These should be included in the rare cases in which they are appropriate, such as for a mobile device. For example, countering excessive power consumption should be included where it is important as a security issue, which is relatively rare (it would only occur if there is concern that an attacker could force this excessive power consumption on a device with limited power).

## B. Selecting Combinations of Tools and Techniques

In general, types of tools and techniques should be selected so that when combined they cover the important technical objectives. One simple approach is to ensure that at least one type of tool or technique adequately covers each technical objective (although having multiple types cover each objective is better). Of course, this requires knowing what the different types of tools and techniques are, and what they tend to be good for.

Appendix E identifies different types of tools and techniques, and our estimates of their effectiveness for different technical objectives. Appendix C provides more detail about the different types of tools and techniques. Appendix E identifies 59 different types of tools and techniques (as columns), while Appendix C identifies 57 different types, but this is not a mistake. In a few cases, Appendix E splits the same underlying

tool (as identified in Appendix C) into different tool/technique types because the underlying tool can be used in different ways that produce different results (e.g., *source code weakness analyzer* is separate from *context-configured source code weakness analyzer*). In addition, Appendix C briefly discusses *formal methods/correct by construction*; in practice these are development processes as well as evaluation processes, and thus they are not within the scope of Appendix E.

Some key aspects of the system affect which tools and techniques can be used. One is information availability: in particular, do you have (1) enough of the source code to modify and rebuild it, (2) source code (not necessarily enough to rebuild), or (3) a binary to examine? Many tools require enough source code to modify and rebuild the software. Another issue is the programming language(s) used; tools are not necessarily available for the language(s) used.

There are many ways to combine types of tools and techniques, and much depends on the technical objectives to be met, the type of software being considered, and the software component's context. We suggest that programs without more experience consider applying at least the following (roughly in order of execution):

1.  Appropriate inexpensive tools and techniques. While many have limited effectiveness, their low cost often makes them attractive, and all of them can counter some potential vulnerabilities. These include:

    a.  Simple attack modeling. Note that attack modeling can be done in far more depth (and be more costly), but simple models of high-level designs can be done quickly and help identify potential sources of problems.

    b.  Applying warning flags. Warning flags cost little to add initially, but can be expensive to add later to an existing project (since repairing reported problems later on can be expensive). If the software already exists, it is still possible to add warning flags later, but this requires adding them slowly and typically requires having a good automated test suite (to ensure that errors are not introduced with the changes).

    c.  Traditional virus scanners. These can find only known simple patterns, but they are very cheap to apply and can counter some trivial attacks.

    d.  Hardening tools/scripts.

2.  Safer languages. If the application is "green field," then it will have more freedom to select the programming language(s) to use. The programming languages C and C++ are valuable when the application must directly interact with hardware or must have high performance, but they are not memory-safe or type-safe. Therefore, they do not automatically protect against certain common errors (e.g., buffer overflows and format string attacks). Many other languages

are memory-safe and/or type-safe, preventing many problems. Where appropriate, languages that provide automatic protection from common types of vulnerabilities should be preferred.

3. Source code quality analyzers. There is now a preponderance of evidence that higher-quality software (in general) tends to produce more secure software [Woody 2014]. These kinds of tools are often less expensive than some other kinds. Higher-quality code tends to be easier to analyze by other tools and techniques, so quality analyzers can improve their effectiveness.

4. Source code weakness analyzers (where source code is available). Source code weakness analyzers have the advantage of being able to examine the entire code base and thus can find vulnerabilities that dynamic-only tools cannot find.

5. Origin analyzer. Most modern software systems are composed of mostly third-party software components. Therefore, it is important to know when a reused component has a publicly known vulnerability. This must be continuously monitored; a component that has no publicly known vulnerabilities today may have one reported tomorrow.

6. Focused manual spotcheck (e.g., for interface authentication). Performing detailed manual analysis can be expensive, but it can be less expensive if focused on specific areas such as ensuring that the external interface requires authentication where it is needed.

7. Web application scanner (if it includes a server-side web application). Many web application scanners are available, and these can quickly find some kinds of vulnerabilities. If the application is a server-side web application, two variants, "Probe-based attack with tracked flow" and "Track data and control flow," might also be appropriate as an alternative.

8. Fuzz testing (fuzz tester, framework-based fuzz tester, and coverage-guided fuzz tester). Fuzz testing can be useful, especially at first, and especially for systems that are not covered by web application scanners. If there is a widely used framework in place, it may be possible to use a framework-based fuzz tester. Fuzz testing may be used to analyze a specific part of the system (the external interface) instead of the entire set of software. The newer coverage-guided fuzz testing tools show great promise for improving the depth of analysis.

9. Negative testing (include tests that are supposed to fail due to security mechanisms properly working). All systems should have an automated test suite; however, many test developers forget to include tests that should fail. Negative tests can quickly address some technical objectives for authentication.

Apple's "goto fail; goto fail" vulnerability is an example of an important vulnerability that could have been caught by negative testing [Wheeler2016].

10. Test coverage analyzer. Software systems should include an automated test suite with good coverage of their custom components. Not only can this detect problems in the custom code, but it also enables rapid update of third-party components when a vulnerability has been discovered (since the automated test suite can be rerun with high test coverage).

11. Digital signature verification. It's important to ensure that the installed software is what was sent; digital signature verification is a relatively inexpensive way to ensure this.

Obviously, this is not a complete list, so projects should consider tailoring this. Different projects can and should make a different selection of tools and techniques, depending on their needs. Systems requiring high assurance, for example, would need more tools and techniques. All of the types of tools and techniques we have identified have their place. Many newer hybrid tools have great promise; we simply lack information on their effectiveness to put them in this list. For examples of selecting combinations of tools, see Chapter 9 (vignettes).

Projects will need to select specific tools of their desired types to address their technical objectives based on the software context (described in Chapter 6). Tool selection depends on other factors than software context, including cost, time, and required level of expertise. It is important to acquire tools and tool licenses consistent with their intended use. In particular, will the tool be used directly by developers or project testers, or will it be used by third-party auditors/evaluators? Some tools or tool licenses are intended for only developers, or only third-party auditors/evaluators, and may be difficult to repurpose. Also, some tools require Internet access and/or uploading of software source code to an external party; that may be inappropriate for some projects.

In some cases, it may not be possible to achieve the desired confidence. For example, if a component is binary-only or services-only, and the suppliers' trustworthiness is uncertain, it may be impractical to use tools and techniques to manage the risks. In those cases, other approaches for managing risks (e.g., by improving transparency) may be necessary.

As discussed in section 5.D, it is often useful to combine multiple types of tools and techniques, and in many cases it is useful to combine multiple tools of the same tool type. Almost all tools have many false negatives (missed reports). Thus, applying multiple tools and techniques is more helpful. Each tool or technique contributes to meeting technical objectives (and thus reducing overall risk).

Existing projects will normally not want to add all tools at once, and certainly not at their maximum settings for detecting problems. Instead, existing projects may select a larger set, but it is usually more practical to gradually add tools and techniques, beginning with relatively easy settings.

# 9.   Vignettes

This chapter briefly illustrates the first steps of the process described in this paper. These steps are identifying the software component context, selecting technical objectives based on that context, and then selecting tool/technique types to address those technical objectives.

The vignettes are based on examples drawn from OTS proprietary software, OTS open source software, custom software, and OTS mobile applications. They omit many details in order to focus on the overall approach. They also intentionally have short lists of technical objectives; many projects might have more objectives, but this would typically require the use of more tool/technique types to counter and would obscure the vignette. A specific program might make different choices based upon its unique circumstances; the purpose of this section is to briefly illustrate the process.

## A.  OTS Proprietary Software Component

The context of this vignette is a program considering the use of OTS proprietary software for which source code is not available. The fact that source code is not available is important because some tool/technique types require access to the source code. (Note that source code *is* available for some proprietary software.) We will assume we do have the software's bytecode as a .class file (e.g., perhaps it is in Java). For the purpose of this vignette, the component is not a critical component and is part of a server-side web application that is accessible through a network interface. We will focus here on identifying the component's technical objectives, and then the tools/techniques to meet those objectives.

### 1.  Technical Objectives for OTS Proprietary Vignette

Given this context, we must identify the technical objectives based on the larger set given in Chapter 4. We apply the approach described in section 8.A, where we first:

1. Consider what kind of component it is. This component is part of a server-side application, so we should consider adding all rows applicable to one as identified in the appropriate "filter for context."

2. Identify the most common kinds of vulnerabilities that apply to this software, and add countering them to the list of technical objectives. Let us assume that the project does not have extensive data about its previous vulnerabilities. In that case, since this is a web application, the OWASP top 10 is one of the more

applicable lists of common kinds of vulnerabilities. This will emphasize some technical objectives, such as SQL injections (which are part of the larger objective to counter unintentional-like weaknesses).

3. Examine the 10 technical objective categories to determine which (other) areas matter for that system's purposes.

When examining the 10 topmost technical objective categories, we consider the following:

1. Provide design and code quality. This is desirable, but given a proprietary component, we will often not have the information necessary for tools to meet this technical objective. We might choose to assume that this will be met based on the supplier's reputation, instead of trying to use tools or techniques to determine this. Thus, we will not include this technical objective.

2. Counter known unintentional-like vulnerabilities. We know these are common attack vectors, so this is likely to be useful. We will include this technical objective.

3. Ensure authentication and access control. We will assume this is merely a component of a larger system that addresses these, so we will not include this technical objective.

4. Counter unintentional-"like" weaknesses. We will assume that the component is not C or C++ (it is provided as a .class file). However, since it interacts with a database, again this suggests that we should at least include countering SQL injection, which is part of this technical objective.

5. Counter intentional-"like"/malicious logic. We do not want malicious logic in the code. Ideally, we would counter any such malicious logic, but we know that trying to address it in all cases is costly. We could decide from the supplier's reputation that the supplier is unlikely to intentionally include malicious code; however, the supplier's development and delivery process may be sufficiently sloppy to allow known viruses into the delivered software. Thus, we may choose to include just the subset of this technical objective for dealing with known malicious software (that has somehow gotten into the component).

6. Provide anti-tamper and ensure transparency. This project has no special anti-tamper or transparency requirements, so we will not include this technical objective.

7. Counter development-tool-inserted weaknesses. Development tools can themselves insert weaknesses. However, we are not the developers, and it is difficult to counter these attacks as non-developers. Thus, we will not include this technical objective.

8. Provide secure delivery.  This should normally be included, so we will include this technical objective.

9. Provide secure configuration.  This should normally be included, however, we will assume that this particular component has no configuration to perform.  Thus, we will not include this technical objective.

10. Other.  These are not relevant in this case, particularly because it is not a critical component.  Thus, we will not include this technical objective.

The last step is to prioritize the technical objectives, in consultation with all stakeholders.  This is not a critical component, so it is more justifiable to drop some technical objectives after considering cost/risk tradeoffs.

After prioritization and considering cost/risk tradeoffs, we chose the following as our final technical objectives:

- *Counter known unintentional-like vulnerabilities*.

- *Counter unintentional-like weaknesses.* For our purpose, countering unintentional-like weaknesses will be met if we select tools to address a majority, if not all, of its relevant subcategories.  For example, we need to address SQL injection.

- *Provide secure delivery*.

- *Counter intentional-"like"/malicious logic with the subset for known malware* (esp. known viruses).  Ideally, we would counter unknown malware also, but that would require much more effort, and so we intentionally limit our objectives.

## 2. Tool/technique Types for OTS Proprietary Vignette

We must now select the types of tools/techniques to meet the technical objectives in this vignette.  Tools that require source code should probably not be considered, since no source code is available.

We first review the types of tools and techniques suggested in section 8.B, with an eye toward covering the technical objectives we identified in section 9.A.1; the ones we select are bolded.

1. Appropriate inexpensive tools and techniques.

   a. Simple attack modeling. Attack modeling could be applied to the larger system that the component will be part of, but it is more challenging to apply it to the component itself built by someone else, so in this vignette we choose to not do this.

b.  Applying warning flags.  Warning flags typically cannot be changed in OTS proprietary components, so we will not apply this.

c.  **Traditional virus scanners**.  These can find only known simple patterns, but they are very cheap to apply and can counter some trivial attacks, so we will use one.  This will help us meet the technical objective for countering "intentional-"like"/malicious logic" in the subset for known malware.

d.  Hardening tools/scripts.  If there is a pre-existing hardening tool that would apply, such as a Security Technical Implementation Guide (STIG), this could be useful. However, for our vignette, we will assume there is no specific hardening tool or script, and we choose to not create one.

2.  Safer languages.  We cannot choose the language for a pre-existing component, so we will not use this.

3.  Source code quality analyzers.  We do not have the source code, so this does not apply.

4.  Source code weakness analyzers. We do not have the source code, so this does not apply.

5.  **Origin analyzer**.  We could choose to use an origin analyzer that works on a .class file.  This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

6.  Focused manual spotcheck (e.g., for interface authentication).  We do not have the source code, so this would be extremely expensive to do and typically would not be worth it, especially since this is not a critical component.

7.  **Web application scanner**.  This component can be executed as a server-side web application, making web application scanners useful, so we choose one. This will help us meet the technical objective "counter unintentional-like weaknesses."

8.  Fuzz testing.  Web application scanners typically include fuzz testing-like functionality, so there is less need for a separate fuzz testing tool.

9.  Negative testing.  Since we do not build the proprietary component, we typically would not have a test suite.  It would be possible to build one, but for a non-critical component we can choose to not do so.

10. Test coverage analyzer.  We do not have the source code, so this is more difficult to do.

11. **Digital signature verification**.  We can do this, to counter attacks during delivery.  This will help us meet the technical objective "provide secure delivery."

After reviewing the initial list, we determined that there are some additional types of tools/techniques to more fully cover the technical objectives:

1. **Binary static analyzers** will be added to increase our ability to meet the technical objective "counter unintentional-like weaknesses" and the sub-technical objectives within it. We have already selected web application scanners, which are a dynamic approach and can miss many vulnerabilities. Adding this static approach could help find what the other tools miss.

2. **Vulnerability scanner** will be added to help "counter known unintentional-like vulnerabilities." Again, this is a dynamic approach that may help bolster origin analysis (a static approach), potentially finding problems other tools miss.

As a result, we have selected six types of tools/techniques to cover four technical objectives; in several cases we have intentionally selected tools to cover technical objectives multiple times. This is not a mistake; most tools/techniques miss many problems, so using multiple types of tools will increase the number of vulnerabilities detected and countered before deployment.

Of course, this is just an example; other tool/technique types could be used in addition or instead (e.g., network sniffers could be used to monitor execution for a period of time to try to detect unexpected "phone home" functionality). We would then select specific tools that implement these tool types and meet our objectives (possibly adjusting the set of tool types as we learn more), and later apply these tools and report results as the project unfolds.

## B. OTS Open Source Software Component

The context of this vignette is that the program is considering the use of an OTS open source software (OSS) component. We assume that source code is available (Java in this case), and this is important since some tool/technique types require source code. To simplify comparison, we will assume that it is essentially the same as in section 9.A: That is, the component is not a critical component and is part of a server-side web application that is accessible through a network interface.

### 1. Technical Objectives for OTS OSS Vignette

Since its purpose and environment are the same, the technical objectives can end up being identical to those in section 9.A.1:

- *Counter known unintentional-like vulnerabilities.*

- *Counter unintentional-like weaknesses.* For our purpose, countering unintentional-like weaknesses will be met if we select tools to address a

majority, if not all, of its relevant subcategories. For example, we need to address SQL injection.

- *Provide secure delivery*.

- *Counter intentional-"like"/malicious logic with the subset for known malware.* Ideally, we would counter unknown malware also, but that would require much more effort, and so we intentionally limit our objectives.

This raises an important illustrative point: technical objectives are essentially a specific kind of requirement, and thus, are not typically impacted by the licensing approach, origin of the software, or information available (e.g., whether or not the customer receives source code). Tools for analyzing the software, however, may very well be impacted by this.

## 2. Tool/technique Types for OTS OSS Vignette

We must now select the types of tools/techniques needed to meet the technical objectives in this vignette. Any of the tool/technique types that we selected in section 9.A are applicable because we have a similar context and the same objectives. The availability of source code, however, opens up the choice of additional tools. Using such additional tool/technique types can provide significant information, but it can be a challenge as well. Source code analyzers often produce a great deal of information that may or may not be relevant to a specific need; extracting relevant information can be time-consuming. Another problem is that it can be difficult to compare the results using source code to another component whose source code is not available. Additional transparency (through source code) can be helpful because it allows many assertions to be verified and problems to be found. However, additional transparency does not make it clear whether the alternatives with less transparency are better or worse.

We first review the types of tools and techniques suggested in section 8.B, with an eye toward covering the technical objectives we identified in section 9.A.1; the ones we select are bolded.

1. Appropriate inexpensive tools and techniques.

    a. Simple attack modeling. Attack modeling could be applied to the larger system the component will be part of, but it is more challenging to apply it to the component itself built by someone else, so in this vignette we will choose to not do this.

    b. **Applying warning flags**. Warning flags typically can be changed in OSS components, so unlike the previous proprietary tool case, we can apply this – and in this case we choose to do so. Note that adding warning flags can be difficult because the software may produce a large number of warnings

when new flags are added.  But in other cases, it is not a problem or those messages can be reported back for repair.  This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

    c. **Traditional virus scanners**.  These are designed to find simple patterns in lower-level code, and they are unlikely to find problems when source code is available.  However, they are so cheap to apply that it does not hurt to use them.  This will help us meet the technical objective for countering "intentional-"like"/malicious logic" in the subset for known malware.

    d. Hardening tools/scripts.  If there is a pre-existing hardening tool (e.g., a STIG) that would apply, this could be useful. However, for our vignette we will assume there is no specific tool or script, and we choose to not create one.

2. Safer languages.  We cannot choose the language for a pre-existing component, so we will not use this.  We have the source code, but rewriting code to another language is a major undertaking.

3. **Source code quality analyzers**.  We have the source code, so this could apply, and we will use it in this case.  This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

4. **Source code weakness analyzers**. We have the source code, so this could apply, and we will use it in this case.  This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

5. **Origin analyzer**.  We could choose to use an origin analyzer that works on our source files.  This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

6. Focused manual spotcheck (e.g., for interface authentication).  We do not have the source code, so this would be extremely expensive to do and typically would not be worth it, especially since this is not a critical component.

7. **Web application scanner**.  This component can be executed as a server-side web application, making web application scanners useful, so we will choose one.  This will help us meet the technical objective "counter unintentional-like weaknesses."

8. Fuzz testing.  Web application scanners typically include fuzz testing-like functionality, so there is less need for a separate fuzz testing tool.

9. Negative testing.  OSS comes with its source code, and typically with a test suite.  It would be possible to add negative tests to that test suite (where available), but for a non-critical component we can choose to not do so.

10. Test coverage analyzer. OSS comes with its source code, and typically with a test suite. It would be possible to add or use a test coverage analyzer, but for a non-critical component we can choose to not do so.

11. **Digital signature verification**. We can do this, to counter attacks during delivery. This will help us meet the technical objective "provide secure delivery."

After reviewing the initial list, we determine that there are some additional types of tools/techniques to more fully cover the technical objectives. There is no need (unlike the proprietary OTS case) to add a binary static analyzer, since we have source code available. However, we might choose other tools, such as:

1. **Vulnerability scanner** will be added to help "counter known unintentional-like vulnerabilities." Again, this is a dynamic approach that may help bolster origin analysis (a static approach), potentially finding problems other tools miss.

In the approach listed here, many more tools are applied to counter unintentional-like weaknesses, because with source code available, more tools can be brought to bear.

As a result, we have selected eight types of tools/techniques to cover four technical objectives. As with the previous case, we have intentionally selected tools to cover technical objectives multiple times. Using multiple types of tools will increase the number of vulnerabilities detected and countered before deployment.

Again, this is just an example; other tool/technique types could be used in addition or instead.

## C. Custom Component

The context of this vignette is that the program is considering custom software development. In this vignette, source code is available (and is sufficient to allow rebuilding the software), the developers may be directed to make changes, and they understand the specific intended environment. For the purpose of this vignette, we will make it similar to the previous vignettes; the component is not a critical component and is a server-side web application.

### 1. Technical Objectives for Custom Component Vignette

To simplify comparison, we will start with the list of technical objectives as in section 9.A.1 and modify it:

- *Counter unintentional-like weaknesses.* For our purpose, countering unintentional-like weaknesses will be met if we select tools to address a majority, if not all, of its relevant subcategories. For example, we need to address SQL injection.

- *Provide secure delivery.*

- *Counter Intentional-"like"/malicious logic with the subset for known malware.* Ideally, we would counter unknown malware also, but that would require much more effort, and so we intentionally limit our objectives.

Note that we have intentionally omitted "counter known unintentional-like vulnerabilities"; because it is being custom-developed, there will be no already known vulnerabilities in the software to be created.

We have a different problem with custom components. OTS components must compete with each other, and this can sometimes encourage quality because low-quality components are less likely to be repeatedly used (unless something else, like low cost or vendor lock-in, compensates for this). This does not apply to custom components; there may have been a bidding competition to develop the component, but there is no competitive alternative to this component. Thus, we would probably want to also add at least this technical objective for a custom component. In addition, we could decide that since we were doing custom development, we would also add this as a technical objective. This means we would add the following technical objectives:

- *Provide design and code quality.*

- *Counter development tool inserted weakness.*


## 2. Tool/technique Types for Custom Component Vignette

We must now select the types of tools/techniques to meet the technical objectives in this vignette. All the above tools can be used in sections 9.A and 9.B, but now analysis results can be tailored for the specific environment. It is easier to direct change in the software, and developers understand the specific intended environment (making some manual techniques easier to apply).

We first review the types of tools and techniques suggested in section 8.B, with an eye toward covering the technical objectives we identified in section 9.A.1; the ones we select are bolded.

1. Appropriate inexpensive tools and techniques.

   a. **Simple attack modeling**. Attack modeling is easier to apply for custom components, and it can quickly warn of design issues that could be costly to fix later. It does not directly guarantee meeting any particular technical objective, but it can help implement many of them; so it could be a sensible technique to use in this case. We would use the attack modeling to help identify all interfaces (information that will be used later).

b. **Applying warning flags**. Warning flags are easy to add initially during custom development, and can be hard to add later; so adding them immediately is sensible. This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

c. **Traditional virus scanners**. It is unlikely to find these problems when custom code is developed. However, they are so cheap to apply that it doesn't hurt to use them. This will help us meet the technical objective for countering "intentional-"like"/malicious logic" in the subset for known malware.

d. **Hardening tools/scripts**. Typically, custom code would be pre-hardened for its purpose, so this would not normally apply.

2. **Safer languages**. We can choose the language for a custom component, so we will use this. In particular, we will avoid languages that are not type-safe or memory-safe when there is no particular reason to use them. This will help us meet the technical objective "counter known unintentional-like vulnerabilities," particularly those involving buffer overflow.

3. **Source code quality analyzers**. We have the source code, so this could apply, and we will use it in this case. This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

4. **Source code weakness analyzers**. We have the source code, so this could apply, and we will use it in this case. This will help us meet the technical objective "counter known unintentional-like vulnerabilities."

5. **Origin analyzer**. An origin analyzer will not really make sense (directly) for the custom software. It would make sense to apply it to any reused software that the custom code uses, but we will treat that separately.

6. **Focused manual spotcheck** (e.g., for interface authentication). We can do a spotcheck, e.g., to ensure that all interfaces do input validation and require any necessary authentication and authorization. Note that the attack modeling could help identify the interfaces. This will help us meet the technical objective "counter known unintentional-like vulnerabilities," especially those involving input validation.

7. **Web application scanner**. This component can be executed as a server-side web application, making web application scanners useful, so we will choose one. This will help us meet the technical objective "counter unintentional-like weaknesses."

8. **Fuzz testing.** Web application scanners typically include fuzz testing-like functionality, so there is less need for a separate fuzz testing tool.

9. **Negative testing**.  Since the custom component is not being tested and used in a variety of other settings, it would be prudent to include negative tests in its test suite, even if the component is not critical.  This will help us meet the technical objective "counter unintentional-like weaknesses." In particular, it can help counter SQL injections (a common concern).

10. **Test coverage analyzer**.  All components need some tests, and it is easy to do very poor testing without a test coverage analyzer.  This will help us meet the technical objective "provide design and code quality."

11. **Digital signature verification**.  We can do this, to counter attacks during delivery.  This will help us meet the technical objective "provide secure delivery."

After reviewing the initial list, we determined that there are some additional types of tools/techniques to more fully cover the technical objectives.  There is no need (unlike the proprietary OTS case) to add a binary static analyzer, since we have source code available.  However, we might choose other tools, such as:

12. **Vulnerability scanner** will be added to help "counter known unintentional-like vulnerabilities."  Again, this is a dynamic approach that may help bolster origin analysis (a static approach), potentially finding problems other tools miss.

13. **Rebuild and compare** will help us meet the technical objective "counter development tool inserted weakness."  This is by no means a foolproof countermeasure, but it can help in some circumstances.

In the approach listed here, many more tools are applied to counter unintentional-like weaknesses, because with source code available, more tools can be brought to bear.

As a result, we have selected 16 types of tools/techniques to cover 5 technical objectives.  As with the previous case, we have intentionally selected tools to cover technical objectives multiple times.  Using multiple types of tools will increase the number of vulnerabilities detected and countered before deployment.

Again, this is just an example; other tool/technique types could be used in addition or instead.  For example, we could add network sniffers (to monitor execution for a period of time to try to detect unexpected "phone home" functionality).

# 10.   Gaps

Our investigation found a number of gaps in analysis tools and techniques that require further research and investment:

- Finding unknown malicious code.  Traditional virus scanners can find many known patterns (although viruses with metamorphic code are much harder to detect).  Unknown malicious code is difficult to find, since by definition there is less certainty about the patterns used by yet-unseen attacks.  This difficulty is further escalated by the large size and rapid change of operational software systems.  Even if every tool/technique type identified in this paper were applied, there would be poor coverage of certain kinds of unknown malicious code (as shown in the "best applicability" column of Appendix E, Software SOAR Matrix).  There are research efforts and approaches for improving this situation, such as work to programmatically predict future malicious code evolutions given existing malicious code, but currently this is a major challenge.

- Integrating different tool results.  It is difficult to integrate different types of tools (e.g., static and dynamic tools), because the kind of information they report is fundamentally different.[10]  Integrating tools is valuable because different tools can identify different issues, so combining them should provide a broader understanding. Standardizing tool output, to enable correlation and synthesis, could help.  SwA correlation tools now exist to help integrate tool results, including information from static and dynamic tools, but more work is needed to improve the correlation tool results.

- Obtaining quantitative data on tools and techniques.  There is a general lack of relevant quantitative data about the true costs, schedule impact, and effectiveness (in various situations) of specific tools, specific techniques, and types of tools/techniques.  A key aspect is inadequate "ground truth" information to help make decisions (e.g., what is the *actual* assurance provided by an industry-accepted set of metrics?).  This lack of quantitative data makes selecting tool/technique types, and selecting specific tools, much more difficult.  There are some ongoing efforts to quantitatively evaluate tools and obtain some

---

[10] Static analysis tools typically report a sequence of one or more locations in code (be it source, bytecode, or binary).  Dynamic analysis tools typically report behavior, e.g., noting that some given input produced a specific output.  Correlating locations in code with behavioral results can be difficult in larger programs.

semblance of ground truth; the SAMATE and NSA CAS have in particular worked in these areas. However, more resources are needed to extend and scale this work.

- Similarly, it would valuable to verify relevant measures of security. Some measures that were mentioned in our interviews included[11]:

    - "Defect density," which is the number of (discovered) vulnerabilities divided by the software size (measured in lines of code or function points);

    - "Technical debt," which can be defined as the expected number of hours needed to repair an identified issue;

    - "Effort density," which is the technical debt divided by the software size (measured in lines of code or function points);

    However, these measures have the following known issues:

    - Tools typically generate false positives. All of the measures above depend on identifying defects or issues to be repaired; a false positive would make relevant measures larger than their true values. False positives could be manually filtered out, but there is an additional cost for doing so.

    - Tools typically generate false negatives. This can be partly countered by using multiple tools, but correlating tool results can be difficult, and this correlation requires effort.

    - Effort estimations can vary as well.

    - Different tools (or tool sets) typically produce different results. This can be partly addressed by using the same tool for a specific decision, but this risks locking into a single vendor's tool over time.

- Including contract language in contracts for assurance. Sample contract language is available that acquisition organizations can choose to insert into contracts [SwAForum 2012] [Marien 2016]. However, unless contracts actually include assurance requirements, assurance is unlikely to be delivered.

- Clear legal authority for analyzing proprietary executables for assurance and compliance. Some analysis approaches can be viewed as performing a kind of reverse-engineering, yet licenses for OTS proprietary executables often forbid reverse-engineering. As a result, some interviewees were uncertain whether they could analyze proprietary executables for the purpose of assurance or

---

[11] There are other relevant measures in the literature as well. For example, the size of the "attack surface" (the set of ways in which an adversary can enter the system and potentially cause damage) has been identified as potentially useful [Manadhata 2008].

compliance.  Suppliers of OTS proprietary executables are understandably concerned about allowing such work, e.g., it might reveal trade secrets.  It might be useful to develop a legal ruling that organizations can analyze proprietary executables for assurance and compliance when the executables have been acquired legally.  Note that attackers already examine proprietary executables to look for vulnerabilities.

- Enabling OTS suppliers to attest to assurance-related activities.  This attestation information could then be used as acceptance or preference criteria.   Currently many proprietary software suppliers are unwilling to provide source code or details of their test process, making many claims difficult to verify.  Even when such data is available, it can be overwhelming to evaluators.  An improved industry consensus-driven set of criteria for certification, backed by a method to verify that the criteria are met, could help to resolve this problem.  It should be possible to enable suppliers to easily attest the analysis they've done in a way that customers can trust (as opposed to simple self-assertion by suppliers), beyond what exists today.  This may require standardization and/or the use of trusted third parties.  An industry consensus-driven set of criteria for attestation could help consumers verify that products meet their criteria.

- Supporting dynamic language static analysis.  Many current languages (JavaScript, Python, PHP, etc.) do not use statically typed variables.  Since less information is captured in the source code, static analysis tools have more difficulty performing analysis, typically resulting in analysis gaps.

- Supporting frameworks. Software frameworks (such as Struts and Spring) can simplify development. Since software security depends on the framework's code and configuration, effective analysis tools must often build in knowledge of the framework's behavior.  The large number of different frameworks being leveraged and reused (especially on servers) requires analysis tool authors to select which frameworks to support, resulting in a lack of support for many frameworks.

- Analyzing binaries without debug symbols. Programs are often created by compiling source code into executable files called "binaries."  These binary files may include "debug symbols" that provide additional information about the program.  Such debug symbols are often very helpful for later analysis programs.  Unfortunately, binaries are often distributed to users without debug symbols, so analysis programs that depend on debug symbols cannot work as well (or at all) on such programs.  Research could be done to improve program analysis capabilities when debug symbols are not available.

- Handling multiple languages and executables. Large systems are usually heterogeneous, with multiple languages and multiple executables. Many analysis tools struggle with such systems because they focus on a smaller set of languages or have difficulty handling systems with multiple executables. Supporting a large set of languages is always a challenge, especially since new ones are developed while legacy languages linger. Systems often include multiple interacting executables, yet many analysis tools are only designed to effectively examine one program at a time.

- Assuring development tools. Development, test, and sustainment tools (including their various plug-ins) can insert unintentional or malicious vulnerabilities into operational software. These include integrated development environments (IDEs), version control systems, compilers, interpreters, test frameworks, and so on. There is some, but relatively little, past work on countering attacks through these tools. One partial countermeasure is to use reproducible builds; the "reproducible builds" website[12] defines them as "a set of software development practices which create a verifiable path from human readable source code to the binary code used by computers." The diverse double-compiling (DDC) technique is a known technique for countering the "trusting trust" attack in which compilers attack software including themselves [Wheeler 2009].

- Cost-effectively and completely meeting a given technical objective without exception (e.g., finding all important vulnerabilities, and not just a subset). Many projects use source code weakness analyzers to identify when source code meets certain patterns that suggest important weaknesses. Yet studies have found that such tools do not detect a majority of vulnerabilities [CAS 2012]. Thus, many tools have a large false negative rate. There are various reasons for this. One is that many tool suppliers are far more concerned about false positives than false negatives; that is, the commercial world rewards them for only reporting true vulnerabilities, even if other vulnerabilities are not reported. Another reason is that the tools often lack important context information required for accurate analysis, e.g., exactly which data sources are trusted.

- Further improving false positive rates in static analysis tools. Tool suppliers are incentivized to reduce false positives, but there is still room for improvement.

- Improving tools reports to be immediately understood and actionable. Tool developers do attempt create reports that help the developer fix potential problems, e.g., they may attempt to prioritize vulnerabilities, identify specific

---

[12] https://reproducible-builds.org/

locations, and/or describe specifically how to fix or mitigate the vulnerability. However, there are still indications that some organizations have difficulty some using tool reports. Thus, it would be valuable to further improve tool reports, especially to ensure that they provide immediately-actionable information on how to fix or mitigate problems they find.

These are not the only general gaps that exist; these are merely more important ones that we identified in the course of this study.

In the mobile environment, we identified the following additional challenges and gaps:

- Brief analysis time. There is a widespread expectation that mobile applications must be evaluated for security within an extremely short time; in many cases these expectations are measured in minutes, not hours, days, or months[13]. This timeframe can apply to either the time between when an application is ready for deployment and its release on an app store, or the timeframe between when a user requests the application for their work device and it is available for installation. The causes of this expectation are unclear; one reason may be that mobile applications are often updated rapidly, making the results of longer evaluation times less useful. This is a challenge if seeking in-depth analysis, especially since in many cases source code is unavailable and an objective is to identify unknown malicious code.

- Many organizations only provide mobile application analysis using a Software-as-a-Service (SaaS) model. In this case, any software to be analyzed must be sent to the external party (the service provider) for analysis. Such services may be inappropriate to use if the software to be evaluated (or the data that must be used with it) must never be available to the public (e.g., because it is classified, proprietary, or specially protected by privacy laws). Additionally, it's difficult to determine or verify what a SaaS-only supplier actually does. Some SaaS suppliers may be very capable, but it is difficult to evaluate them due to this lack of information. (This not only impacts potential service users, but it also impact this paper, as it is difficult to comment on services when there is little data available about them.) See Appendix B.1 for more information about data availability and its impact on using tools.

- It is sometimes difficult to determine how to characterize some tools. As noted above, tools are rapidly evolving. In addition, sometimes information is difficult

---

[13] On Android devices there can be many app stores; applications tend to be rapidly available on the Google Play store. On Apple iOS devices there is a lengthier time between when an application is submitted and when it is available on the store, but it is still short compared to traditional analysis timeframes. For more information, see Appendix F.

to get, especially when the tool is only available through SaaS.  We have tried to identify major approaches used by tools to divide them into families, in cases where we could obtain more information.  However, some tools combine a number of approaches that make them challenging to categorize.  It may be that in some cases it would be better to create specialized columns for specific tools, instead of looking for general categories.  Examples of tools that were especially difficult to categorize are Veracode's vAI and Kryptowire:

- Vericode's vAI includes both static and dynamic analysis.  In particular, it identifies a number of red flags and then uses machine learning techniques to estimate risk level.  Thus, it uses a large number of different small simple analysis techniques, instead of a single primary technique.  This tool could be viewed as focusing on a different approach for combining data, instead of a different technique for obtaining this data.  This was difficult to map using our tool/technique family; we could have created a new hybrid category but we believed this would not fully capture the approach.

- Kryptowire also applies several different analysis approaches and combines their results.  These analyses are primarily dynamic, but some are static (e.g., it identifies libraries and their dependencies).  In addition, the analysis approach it uses to analyze Android applications is fundamentally different than the one it uses for iOS applications.  On Android they translate to bytecode and force execution through different paths (bypassing conditions in code where necessary), an approach we call "forced path execution."  On iOS they take the unencrypted application and exercise it on a modified iOS (to see what paths it takes).

- Devices that support mobile communications capability (e.g., 3G or Long-Term Evolution (LTE)) typically use a separate baseband processor running a separate real-time operating system (RTOS) and programs that manage everything related to the radio and often other capabilities as well (e.g., Global Positioning System (GPS) and Universal Serial Bus (USB)).  The baseband processors and associated software are typically poorly understood, poorly documented, and not externally peer reviewed.  This leaves mobile devices exposed to over-the-air attacks that may enable total control of the device, yet these attacks may be poorly countered or mitigated.  [Holwerda 2013].  See Appendix F for additional information.

- It is difficult for users, including those testers who operate as regular users, to understand what is occurring on their mobile devices.  Applications routinely communicate with each other, but this communication is not obvious to typical users.  Similarly, applications often require privileges, but the impact of granting them is often poorly understood.  This lack of understanding increases the risk to

users and their enterprises, since users are less likely to notice unexpected activity on their mobile device.

- Balancing the need between efficiency in operation with separating personal and professional data. Application data is separated from other applications by default on both iOS and Android. In many cases there is a need to share (e.g., calendars need both personal and professional information), but there are also many reasons for separation (e.g., because of privacy, intellectual rights, regulations, preventing unauthorized sharing, etc.). Enterprises want to be able to delete their data from mobile devices, while users understandably do not want their personal information deleted.

These gaps would be plausible areas to consider as part of such a research program.

# 11.  Conclusions

Nearly all modern systems depend on software.  Although software enables functionality, it also poses risks.  Unintentional and intentionally inserted vulnerabilities in software can provide adversaries various avenues through which they can reduce system effectiveness, render systems useless, or even turn our systems against us. Unfortunately, it can be difficult to determine what types of tools and techniques exist for evaluating this software, and where their use is appropriate.  DoD PMs, and their staffs, need information and guidance on how to make effective software assurance (SwA) and software supply chain risk management (SCRM) decisions, particularly when they are developing their program protection plan (PPP).  DoD policymakers who are developing software-related policies also need such information.

We propose the following approach for selecting tool/technique types to address various technical objectives:

1.  Select technical objectives based on context,

2.  Select tool/technique types to address objectives,

3.  Select tools,

4.  Summarize selection (e.g., as part of the PPP),

5.  Apply and report.

For step 1 we have identified a set of common technical objectives.  For step 2 we have identified a set of tool/technique types, as well as a matrix (in Appendix E) that shows how the different tool/technique types address different technical objectives.  Note that the different tool/technique types can be grouped into three larger groups: static analysis, dynamic analysis, and hybrid analysis.  We also provide additional information to support steps 3-5.  We believe that analysis should be performed across the lifecycle, as discussed in Appendix D.

Many gaps exist, as described in Chapter 10.  These include finding unknown malicious code, obtaining quantitative data, analyzing binaries without debug symbols, and assurance of development tools.

In the mobile environment, there are additional challenges.  Many tools are less mature, simply because the mobile environment is newer and evolving, though we expect that to rapidly improve.  There is an expectation of short analysis time constraints that preclude many approaches to in-depth analysis.  Attempting to counter unknown

malicious code with minimal time for analysis is a particular challenge.  In addition, the SaaS model makes it difficult to evaluate an evaluation process' effectiveness, and constrains what software can be evaluated, yet some tools are only available through a SaaS model.  The widespread use of a Software-as-a-Service (SaaS) model as a *sole* evaluation model limits data availability and application to DoD systems

These identified gaps would be plausible areas to consider as part of the research program identified in the National Defense Authorization Act for Fiscal Year 2014 [NDAA 2014] Section 937.

# Appendix A.
# Resources Used

We used a variety of information sources to develop this document, including interviews, conference attendance, and written materials. Below we identify the major sources for the original paper, followed by additional major resources for the mobility work.

For the first draft of this paper, we conducted a large number of interviews with a variety of individuals from diverse organizations. These interviews were typically 1 to 2 hours long, and began with these three questions:

1. What technologies, tools, and techniques would you recommend for verifying software being considered for use? Under what conditions?

2. What is the estimated level of effort, cost, difficulty, gaps, and other practical aspects of using the tools that a prospective user would want/need to know?

3. What are case examples of how/when to use technologies/ tools/techniques (e.g., how often/when to scan)?

Our original interviewees were:

1. Government community (other than DoD Services):

   − National Institute of Standards and Technology (NIST) (Paul Black, Vadim Okun, Aurelien "Aurey" Delaitre) – 2012-12-20;

   − National Security Agency Center for Assured Software (NSA CAS) (Andrew Portner (Binary), Nick Valletta (Mobility), Kathy Erno (Source lead), Rob Stevens (Source), and Jon Spielvogel (Source)) – 2013-01-30;

   − Carnegie Mellon University (CMU) (Robert Seacord) – 2012-10-21;

   − United Kingdon (Ian Bryant) – 2012-10-29;

   − National Information Assurance Partnership (NIAP) (Mark S. Loepker, Director, CCEVS) – 2013-03-11;

   − Office of the Assistant Secretary of Defense for Health Affairs/TRICARE Management Activity (OASD(HA)/TMA) (John Keane) – 2013-02-06.

2. DoD Services:

   − Air Force (MSgt William Tooke, Lt. Booth, TechSgt Adams) – 2012-01-29;

   − Application Software Assurance Center of Excellence (ASACoE);

   − Navy – (Jennifer Guild) – 2013-01-22;

   − Army – (George Huber) – 2013-02-01;

   − Additionally, IDA led a discussion at DoD Software Assurance (SwA) Community of Practice (COP), with all services represented – 2012-09-17.

3. Vendors/Suppliers:

   − Hewlett-Packard/Fortify – (Jacob West, Director, Software Security Research) – 2013-01-08,

   − VeraCode – (Chris Wysopal, CTO/Co-founder) – 2013-01-17;

   − Coverity – (Andy Chou, CTO/Co-founder) – 2013-01-25;

   − Electronic Warfare Associates Information and Infrastructure Technologies (EWA IIT, Steve Clemmons) – 2013-02-13;

   − Juniper (David Koretz) – 2013-02-27;

   − Tenable (Ron Gula) – 2013-02-20;

   − SafeCode/EMC (Dan Reddy) – 2013-03-15;

   − IBM (Andras Szakal, Chan Lim, Diana Kelley, Jim Whitmore, Rustin Sides) – 2013-03-21;

   − McAfee (Phyllis Schneck) – 2013-03-27;

   − John Viega – 2013-03-27;

   − WhiteHat (Jeremiah Grossman, Kyle Rohrs, Philip Diaz) – 2013-04-02.

We are grateful for the time our interviewees gave us. All provided a great deal of insightful commentary; we include some of their observations in Appendix B, but these are only some of their insights due to length constraints. Also, please note that while we used their comments, for the most part none of them have seen this report, and thus they have not approved of this report. We list their names to acknowledge their contribution, not their consensus.

We also participated in a number of conferences and workshops:

• NIST SCRM meeting, October 15–16, 2012, Gaithersburg, Maryland;

• Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) Principle Investigator (PI) meeting, November 14, 2013;

- SwA Working group Sessions – Winter 2012, 27–29 November, McLean, Virginia;

- Annual Computer Security Applications Conference (ACSAC), December 2–7, 2012 Orlando, Florida;

- RSA[1] conference February/March 2013;

- SwA Forum, March 5–7, 2013, NIST, Gaithersburg, Maryland;

- High Confidence Software & Systems (HCSS), May 5–10, 2013.

We also examined a number of written works. Some of the especially-useful ones were:

- Information on Application Software Assurance Center of Excellence (ASACoE)'s process [Tooke 2012],

- DoD key documents, including DoD Instruction 5200.44 [DoDI 5200.44] and the Defense Acquisition Guidebook [DAG],

- SAFECode material, e.g., [SAFECode 2012],

- Open Group's Open Trusted Technology Provider Framework (O-TTPF) [Open Group 2011],

- Booz Allen Hamilton's Software Security Assessment Tools Review [BAH 2009],

- National Security Agency (NSA) Center for Assured Software (CAS)'s reports, e.g., [CAS 2012].

Useful websites included:

- Common Weakness Enumeration (CWE), http://cwe.mitre.org,

- Software Assurance Metrics And Tool Evaluation (SAMATE), http://samate.nist.gov/Main_Page.html,

- Building Security In Maturity Model (BSIMM), http://bsimm.com/.

We later extended the report by looking specifically at mobility issues.

People we interviewed when we were specifically focused on mobility issues included:

- Veracode (Chris Wysapal), 2013-10-28,

- NIST SAMATE (Paul Black), 2013-10-29,

---

[1] RSA stands for (Ron) Rivest – (Adi) Shamir – (Leonard) Adleman, but this is rarely expanded.

- NSF (Jeremy Epstein), 2013-10-30,

- George Mason University (Sam Malek), 2013-10-30,

- NIST mobility (Jeff Voas), 2013-11-01,

- George Mason University (Angelos Stavros), 2013-11-01,

- Symantec (Paul Sangster), 2013-11-12,

- eVault (Wyllys Ingersoll, Security expert and iOS application developer), 2013-11-20,

- Apple (John DiTomasso), 2013-11-21,

- FireEye (Eric O'Brien), 2013-11-22,

- Defense Advanced Research Projects Agency (DARPA) (Tim Fraser), 2013-11-26,

- DARPA (Doran Michels), 2013-12-12.

We also received email comments from David Wagner, a well-known security expert.

We also reviewed a number of documents (including program documentation) related to mobility, particularly those relating to security or government use. These included mobility-related documents from the NSA CAS, Defense Information Systems Agency (DISA) mobility Security Technical Implementation Guides (STIG), Department of Homeland Security (DHS) "carwash" information, information on the DARPA Transformative Apps (TRANSAPPS) program, information on NIST's evaluation process, iOS security documentation, and Android security documentation [DISA STIG 2013], [DARPA 2013], [Walker 2013], [CAS 10x10 2013], [CAS Survey 2013], [CAS Vuln 2013], [Android 2013], [Apple 2012], [NIST SP-163], [NIST SP 800-124rev1 2013], [NIST 2012], [NIST 2013].

Many people observed that different tools and techniques were better for different objectives. In particular, Andy Chou and the NSA CAS emphasized the value of comparing tools and techniques with differing objectives, and this was an inspiration for the development of our matrix. We thank them for their insights, however, please note that they have not reviewed our specific rows, columns, rating system, and entries. The matrix (including its entries) represents our effort to summarize information from a variety of sources, and not from just them. We believe that the matrix entries require further community review, vetting, and sustainment. Nevertheless, we believe they provide a useful starting point.

We are grateful to all who provided us information, including interviewees, those who provided information through conference presentations, and the authors of the various documents we used, and thank them all.

# Appendix B.
# Key Topics Raised in Interviews

Interviewees provided a number of interesting and valuable insights that did not fit anywhere else in this paper. This appendix is a summary of some of those insights, based on our interpretation of their comments. This is only a subset due to length constraints, and it is possible that some interviewee would disagree with a point brought up by others. As agreed prior to the interviews, there is no attribution to any specific individual.

## 1. Key Issue: What Data are Available?

A key issue in tool selection is to determine what data are available for the target of evaluation (TOE). Tools cannot be used if data they require are unavailable. Data availability can be roughly grouped into these categories:

1. Service only (no executable),

2. Executable only (binary/byte code),

3. Source code without build source ("can't build it"),

4. Source code, can rebuild from source (this category is required for many tools),

5. Source code, can rebuild and direct change.

Software in a given program can be divided into "custom" and "off-the-shelf" (OTS) software, which is correlated to but not the same as these categories. Custom software is often in category 5 (the source code is available, it can be rebuilt, and changes can be directed)… but sometimes it is not. OTS software can often be further divided into services (typically category 1), proprietary software (typically category 2), and open source software (typically category 4)… but again, this is not always true.

A key point, however, is that many proprietary commercial OTS (COTS) suppliers will *not* provide source code, or will do so only at a large extra cost. This lack of data means many tools cannot be brought to bear, which could mask more serious problems. It also complicates evaluating software alternatives; if serious problems are found in software where more data are available, while fewer problems are found in software where fewer data are available, this may mean that the latter has fewer problems, but it may instead mean that there are the same or more problems that are masked by the lack of data.

## 2. Organizational Approaches

Different organizations take different approaches to evaluating software, in part because of the impact of the data available, as well as other issues such as threat. For example:

- The Air Force Application Software Assurance Center of Excellence (ASACoE) goes out to developers and trains them, and helps to acquire and install tools in their environment.

- The Software Engineering Institute (SEI) performs some third-party evaluations for others after being given source code. However, it does not necessarily have the build/execution environment (such software is in category 3), which in those cases limits the kinds of analyses SEI can perform.

- One commercial third-party evaluator examines source code, and rebuilds the executables from source to ensure that the same executable is produced. They believe that the ability to rebuild identical executables from source code is critical when looking for potentially malicious unknown attacks; "otherwise [evaluators] can't counter unknown malicious code."

Additionally, it important to note that programs are usually not resourced to do detailed evaluations of all OTS components. Typically, they must focus on only the most critical OTS components.

## 3. Other Comments

There were a variety of other comments as well that are not trivial to organize, but seemed worthwhile to record. These comments are listed below.

1. **No silver bullet.** There was general agreement that there is "no silver bullet" for security issues. Analysis requires hard work, and it can never provide a "100%" guarantee." That said, there was general agreement that the technology exists to improve current practices.

2. **Training gaps continue.** "Training gaps" were repeatedly mentioned. Training is vital for developers, analysts, and tool users. Most software developers still receive no education or training in how to develop secure software, leading to the large number of easily exploited unintentional vulnerabilities in much of today's software. Although tools can help identify or remediate problems, automated tools by themselves cannot catch all vulnerabilities, they typically require significant training and ramp-up time for developers to gain the necessary expertise to use the tools, and often users must understand how to develop secure software in order to use the tools.

3. **Evaluate throughout development.** Early and repeated evaluation during development is far better than evaluation after a system has been completely developed. In particular, enabling rapid feedback is critical to minimizing cost and schedule impact. One interviewee argued that, "if notification occurs within a week of development, it typically takes less than 15 minutes to fix." After 3 months or more the developer has forgotten many details; this means that fixing any problem takes longer, and if the notification occurs late in the development, schedule pressures can be an exacerbating problem. Also, without early generation of vulnerability reports to help to train developers to avoid the problem, the same problem may be repeated many more times. One organization reported that it was often asked to review software late in the development lifecycle, typically just before deployment as part of the certification and accreditation (C&A) process. It is difficult to address problems at that point; schedule pressures provide little time to look for problems, and it is difficult to fix the problems that are found. Cost-effective analysis is facilitated by integrating at least some tools into the development/sustainment environment, e.g., in the software's integrated development environment (IDE) or configuration management (CM) system. It is best to incrementally increase tool use and acceptance thresholds, using feedback to determine where to increase the thresholds next.

4. **Make SwA a requirement.** Several interviewees reported that DoD programs often don't consider SwA a requirement. As a result, they do not prioritize the funding of assurance efforts, including the purchase of analysis tools, use of assurance tools and techniques, and making changes based on their results. Program managers often presume assurance is the job of information assurance (IA) specialists and fail to understand that assurance is a key part of system readiness. Some program managers plan to "bolt on" assurance later, with predictably poor results.

5. **Contract for assurance.** Instead, government programs need to contract for assurance. One interviewee stated, "there's nothing in the FAR that prevents us from preferring suppliers who can demonstrate that they've done code scanning or will provide source code so we can scan it" – yet this is not currently normal practice. Some interviewees reported that the government often does not receive the source code or unlimited rights to the software that it paid to develop. This lack of data and data rights inhibits analysis of that software, as well as future competition for that software's maintenance.

6. **Higher software quality may improve SwA tool effectiveness.** There is a widespread perception that first using "code quality" tools (which are often simpler or cheaper) and fixing the issues they find makes other tools more

effective. In particular, it is probably easier to analyze cleaner and less complex code, making other analysis tools more effective. Several experienced people have expressed this belief, and we think this is probably true. Unfortunately, we have not been able obtain data to prove this claim. Chapter 9 notes that it would useful to perform experiments to verify this.

7. **Differentiate problem-finding from risk assessment.** It is important to distinguish between those finding problems so that they can be resolved versus those trying to assess overall risk. Different tools focus on different aspects of these goals, e.g., they may perform different trades between false positives and false negatives. Also, different sets of information and expertise are typically required. Software developers, with their detailed understanding of the software they developed, are usually in a better position to fix vulnerabilities. Auditors/selectors may understand the risk and context better for a given system (since they may better understand the larger picture of the mission and what role the software plays in it).

8. **Triaging requires people.** Triaging true positives from tool warnings must, in the end, be done by people. A situation may or may not be a problem depending on the operational environment and mission; for example, an input may appear to be a problem but in fact be impossible to exploit due to larger processes. However, it is important to consider that it is sometimes easier, faster, or less expensive to fix a possible vulnerability than to do the in-depth analysis to determine exploitability.

9. **Summarize and identify what is important.** It is critical to "boil up" complex, lower-level results into simple results that can be communicated and acted upon. One common mistake is to simply summarize the number of potential vulnerabilities, independent of their relative importance. One interviewee recommended that it was better to say "no vulnerabilities worse than X," with multiple levels, or be able to say, "Your security is at least up to some level." This is, of course, challenging to do in practice.

10. **Automation is necessary.** Automation is important for scale, affordability, and reduction of human error. Experts are important, but "experts don't scale." Where practicable, it is best to combine automation and human review.

11. **False positives and false negatives affect tool utility.** Tools are subject to false positives, false negatives, and often both. Users of tools with false positives must typically apply effort to determine whether some report is actually true, in addition to the potential effort of fixing a problem if it is true and worth fixing. In some cases it may be easier to fix a reported problem, even if the problem does not exist, than to determine whether there actually is a problem. Users of

tools with false negatives should be aware that even after using them, residual vulnerabilities probably remain. Tools with false negatives can still be very useful, but it's often important to compensate for their limitations (e.g., by using additional tools or mitigations). Tools can be designed to have low false positive rates or false negative rates, depending on their intended use. As one interviewee put it, "developers hate false positives." However, auditors worry about false negatives. An interesting challenge is that false positives can accumulate in real systems; as "real" defects are found by tools and then fixed, false positives often remain. These accumulating residual false positives can even discourage tool use over time, since they give the misleading perception that tools do not find real vulnerabilities.

12. **Need for static analysis ground truth.** A number of comments were made about static analysis tools in particular. NIST and NSA have both made progress on establishing "ground truth" on the effectiveness of some types of static analysis tools, but this task is hard and will require significant resources and time. NSA CAS's reports show that existing tools in their scope identify a minority of vulnerabilities (at most 25% of the Juliet test suite), with relatively little overlap between tools. This highlights the importance of using multiple static analysis tools to get better coverage, but using multiple tools and combining their results raises costs.

13. **Dynamic analysis limited by execution environment availability.** A number of comments were also made about dynamic analysis tools. Many find them useful, but using them requires an executable environment. This requirement sometimes poses a challenge due to problems in acquiring the software, related data and artifacts, or in assembling the relevant execution environment.

14. **Combine approaches.** Many believe it is best to combine various analysis approaches. Combining static and dynamic approaches helps focus on areas that need further examination (e.g., suspicious, unknown, and critical issues). Also, combing dynamic test approaches with static analysis can allow better identification of test coverage, making it easier to determine what is left untested. Unfortunately, it is often difficult to correlate the results of static and dynamic analyses. Combining automated tools with manual review techniques is another remarked-upon approach. Manual review can be more comprehensive but is typically much more costly. These costs have to be balanced against relying solely on automated tools; one interviewee stated that "tools aren't even getting half of the vulnerabilities" (a view supported by NSA CAS research results).

15. **Tools are only part of the process.** There were some complaints that some tool suppliers promise, or at least imply, more than they can deliver. This is not to say that tools cannot help; tools can help. But users must understand that tools

are limited and understanding their limitations is an important part of using tools. In particular, tools should be part of a larger process for evaluating software, not the only mechanisms in use.

16. **Focus on program-specific issues as driven by context.** There are various "top" vulnerability lists, such as the CWE/SANS top 25 and the OWASP top 10. While these can be helpful in understanding general trends, the most important vulnerabilities for a given program are driven by context. Programs should not naively apply top lists, but instead should use them as inputs and focus on what is important for their context.

17. **Integrate operations and development.** There is often relatively little integration between the operational and developmental environments. For example, developers may receive bug reports, but they receive relatively little information about the kinds of attacks that systems undergo. This lack of data sharing makes it more difficult for developers to develop systems appropriate to operational needs.

18. **Ensure that executables correspond to source code.** Before evaluating source code, it is important to ensure that the executables correspond to the source code so the effort is not wasted since what is examined might not represent what is actually executed. This is especially important if there is a concern about maliciously inserted functionality. The rebuild-and-compare process can determine whether the source and executable correspond (presuming the development tools are not malicious) by separately rebuilding the executable from the source code to compare against the delivered executable. However, this process requires very detailed build environment information and information sharing between the builders of the original executable and the re-builders of the checked executable.

19. **Ensure that the test and operational environments match.** The test development environment must really match the production environment. Vulnerability results often differ between the test environments (such as the quality assurance mirrors) and the production environments. One reason is a developer of off-the-shelf (OTS) software might test their software on "stock" platforms, but not on hardened platforms. Another reason is that some patches for a given platform cannot be operationally deployed – something that all developers, including the patch creators and anyone depending upon the patched system, should be aware of. If not, developers of software that depend on that platform may incorrectly presume that the operational platform will be patched, potentially leading to open vulnerabilities.

20. **Functional testers and security evaluators must communicate.** Those responsible for functional system test and those responsible for software security evaluation must communicate. System tests, for example, can clarify what is exploitable in an environment, which is critical for those evaluating in software security.

21. **Share risk and compromise information.** Where possible, improve the sharing of operational attack and compromise information. Developers need certain kinds of compromise information so they can better identify vulnerabilities, more appropriately respond or mitigate vulnerabilities, and avoid creating similar vulnerabilities in the future. This information is also needed by organizations that identify and implement countermeasures (e.g., configure firewalls and web application firewalls). In general, it was observed that attackers share information, while defenders often don't, because of defender concerns about factors such as liability and intellectual property (IP) loss. One interviewee commented that there is "honor among thieves, but not [among] competitors." DoD understandably classifies a significant amount of information about attacks and compromises. However, many developers are not cleared to the same level or have difficulty accessing this type of classified information. Thus, developers for DoD systems are often unaware of the amount and nature of the attacks that their systems must counter or of current vulnerabilities in the systems they maintain, potentially leading to the development and sustainment of systems unprepared for their operational environment.

22. **Support damage reduction, damage detection, and recovery.** No system can be demonstrated to be invulnerable to all attacks. Modern systems must be resilient, designed to reduce damage, detect problems, and recover from exploitations. As one interviewee put it, "Everybody's owned. Everyone has viruses; not everyone gets sick." Enabling damage reduction and recovery often requires detecting that there is a problem, which suggests that software should be designed with built-in real-time sensors that are always enabled so that problems are monitored in real-time.

23. **Consider earmarking funds for CPI protection.** In some cases the government funds research work where critical program information (CPI) exists, will exist, or is likely to be created (CPI is defined in DoD Instruction 5200.39). Often these programs are funded as Small Business Innovation Research (SBIR) programs. One interviewee proposed that the government consider earmarking some of that research money specifically to prevent unauthorized distribution of CPI.

24. **Firmware analysis is difficult.** Analysis of firmware is especially difficult. Often no source code is available for firmware, and relatively few tools are

designed for it. Additionally, firmware typically depends on low-level details that are not widely available.

25. **Quarantine software.** Some interviewees suggested that software from "outside" should be quarantined, enabling potential users to build up trust. One analogy was that the software should "take a shower before swimming"; that is, the organization should do simple checks of the software, put it into an artifact repository where data from its use can accumulate, and then check it again before using it in a final build.

26. **Protect the development/sustainment environment.** It is important to protect the development/sustainment environment along with the operational environment. This includes preventing attacks from getting in (e.g., for integrity and availability) and preventing important information from getting out (e.g., for confidentiality). This includes protecting against malicious developers, malicious testers, and even malicious administrators of the development environment.

27. **Both centralization and decentralization have benefits.** Arguments exist for both decentralization and centralization of evaluation processes. Decentralization benefits include the potential for broader system and environment expertise. Centralization benefits include the potential for gaining and using tool/ technique expertise and for sharing OTS evaluation resources. A combination of decentralization and centralization may offer the most value, e.g., decentralized evaluation of custom software, backed by more centralized tool/technique expertise when needed.

28. **Identify common defects of similar systems.** Defect types are often prevalent in a given type of system. Thus, it is important to identify the various security-related defects in a given system so that developers of similar systems will look for them and know what not to repeat.

29. **Use code signing.** Code signing by the supplier, for testing by the recipient, should be considered a minimum practice for critical software. It cannot counter all problems, e.g., the software may have vulnerabilities as supplied or it may be misconfigured. Still, code signing can counter a number of software supply chain attacks.

30. **Ensure customer is in control.** Mechanisms in software that allow external control (such as "call back to vendor," automated patch updates, and diagnostic/service interfaces) are of concern to many government organizations. These are not considered backdoors by much of industry, but government customers often want to control communications and changes of the software that they depend on. Acquirers should require all external software interfaces to

be under their control and that the software enables easy, centralized configuration of these controls. Similarly, suppliers should ensure that all external interfaces of the software they supply are under customer control and the software enables centralized configuration of these controls. These software interfaces and configuration should be documented as necessary to ensure that customers are ultimately in control of their systems.

31. **SaaS-only analysis services are sometimes inappropriate to use and are difficult to evaluate.** Some tool suppliers provide vulnerability analysis as an external service, that is, they provide analysis as Software-as-a-Service (SaaS). In many such cases, the services are *only* available as a SaaS. Such services can have advantages for customers, e.g., there is no software or hardware to install (speeding initial use), and the tool supplier can filter and simplify results for more efficient use. They also provide advantages to the tool supplier; the tool can be more easily updated and trade secrets are easier to protect. However, services may be inappropriate to use if the software to be evaluated or its data must never be available to the public (e.g., because it is classified or proprietary). There is also the concern that the service provider may gain insight into system vulnerabilities that could be later exploited by adversaries through a variety of means. For example, the service provider might not tell the customer about all vulnerabilities found, or the list of vulnerabilities found might end up (intentionally or not) in the hands of an adversary. An additional problem is that it's often difficult to determine or verify what a SaaS-only supplier actually does. Many SaaS suppliers do not clearly describe in detail the types of analysis they perform, the tools they use, the accuracy or completeness of the results, and so on. It is often unclear if the SaaS supplier is just reusing existing public tools or is providing valuable capabilities (e.g., their own custom tool/technique or strong specialized expertise).

# Appendix C.
# Fact Sheets

These subsections describe each tool/technique type, providing:

- Overview.  A brief description of the tool/technique type, including common alternate names.

- Details.  A more detailed description of what the tool/technique type is and how it works, including general capabilities.

- Applicability.  Information on when and where it applies (and when it doesn't). This includes the applicable parts of the lifecycle, types of software components it does or doesn't apply to (e.g., applications vs. kernel code, embedded vs. non-embedded), and the data it requires (e.g., if it requires source code).

- Assessment.  A description of what it is good for, in particular, its pros and cons. In the pros and cons the word "it" refers to the tool/technology type unless otherwise noted.

- Resource requirements.  A qualitative description of what it costs including licensing, training, and so on.  Many factors affect cost, so we focus on identifying the key types of costs likely to be relevant.  It is especially important to note that many tools/techniques have one-time costs as well as continuing costs, and the distinction must be understood to properly scope any investment. Nearly all tools and techniques require at least some training costs.

- Examples of suppliers/products.  These examples are intended to help readers further understand the tool/technology type. ***The lists of examples are illustrative, and <u>no</u> endorsement is implied.***

More detailed information about specific tools can, in some cases, be found in [BAH 2009] and [CAS 2012].  Lists or evaluations of specific types of tools and techniques are described in their corresponding fact sheet.  NIST Software Assurance Metrics and Tool Evaluation SAMATE[2] and the OWASP Benchmark Project[3] provide some test suites/benchmarks for testing tools.

---

[2] https://samate.nist.gov/Main_Page.html and https://samate.nist.gov/Other_Test_Collections.html

[3] https://www.owasp.org/index.php/Benchmark

# 1. Attack Modeling

### a. Overview

Attack modeling analyzes the system architecture from an attacker's point of view to find weaknesses or vulnerabilities that should be countered.

### b. Details

Attack modeling is used to analyze the system architecture from an attacker's point of view; examples of attack modeling processes include "threat modeling" approaches [Hernan 2006] [Meier 2003], using attack trees [Salter], developing abuse cases, performing attack surface analysis, and examining attack patterns. Attack modeling may be system-centric, asset-centric, and/or attacker-centric. System-centric approaches focus on the components of a system, examining each component and their interconnections (especially at trust boundaries). Asset-centric approaches focus on the key assets to be protected. Attacker-centric analysis approaches focus on the attacker (e.g., the attacker's goals or methods).

Attack modeling may use the Common Attack Pattern Enumeration and Classification (CAPEC) attack patterns as a list of typical attack patterns to see what the system is vulnerable to. Modelers may choose, for example, to use the "mechanism of attack" (CAPEC-1000) to ensure that they cover a large set of attack methods. See http://capec.mitre.org for more information on CAPEC.

Unlike penetration testing, an analyst using attack modeling does not actually perform the attack. Instead, the analyst analyzes system artifacts (such as designs) to anticipate system vulnerabilities. Thus, attack modeling can be performed before the system is implemented.

### c. Applicability

This approach typically requires architectural information about the system and is best applied before the system is implemented, e.g., as part of the design process. It can be applied after the system is implemented, but at that point it is more difficult to apply many countermeasures (such as changing the architecture), potentially reducing the utility of attack modeling.

### d. Assessment

Pros:

- Attack modeling can be applied before system implementation, allowing early identification and cost-effective resolution of potentially critical vulnerabilities.

- It can be relatively low-cost.

Cons:

- It has limited applicability to the internals of OTS components, since often their implementation cannot be changed or change is more limited.


- It can be difficult to apply, because many developers are unaccustomed to thinking like an attacker.
- Training is necessary.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Training, including costs for training, and the time of those receiving the training.

    − Development of initial analysis, and revising system architecture products to counter identified vulnerabilities (if necessary).

    − Tools, where used.

- Recurring resource requirements:

    − Modifying analysis as architecture changes.


### f. Examples of Suppliers/Products

Attack modeling can be done with simple tools (e.g., whiteboards and office automation tools) depending on the specific approach chosen, especially if very abstract models are used.  A number of tools exist to track more detailed models.

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Microsoft | SDL Threat ModelingTool | http://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx |
| Led by SINTEF | Seamonster | http://sourceforge.net/projects/seamonster/ |

## 2. Warning Flags

### a. Overview

Warning flags are mechanisms built into programming language implementations and platforms that warn of dangerous circumstances. This technique maximally enables relevant available warning flags; developers then resolve problems that the warning flags identify before the system becomes operational.

### b. Details

Many programming language implementations include mechanisms to warn or prevent dangerous circumstances. Since the mechanisms are built into the language implementations themselves, and are applied when they are processing source code, they are immediately available for use. In some cases they are enabled by default (but can be disabled); in many others they must be specifically enabled.

Warning flags are static mechanisms that are fundamentally similar to the source code weakness analysis tools described elsewhere. However, although their analysis tends to be less detailed than source code weakness analyzers, they have the advantages of being built into the compiler or related tools. Since they are built into the compiler or related tools, they have information about what the compiler will really do (a separate tool must model this and its model may be incorrect). In addition, there is no financial impediment of having to acquire something separate and integrate it into the development process, and the tools' analysis can be automatically repeated every time the software changes (not just when some separate analysis tool is run). There is no conflict between these two types of tools, and organizations may choose to both enable warning flags *and* apply source code weakness analyzers.

### c. Applicability

These tools require source code. Thus, the tool user must have access to the source code. Note that the tool need not be a traditional compiler that generates machine code; many scripting language implementations include a process that translates the source code into some intermediate format, and this process can often implement warning flags.

It is far more practical to enable this tool/technique early in development, before code writing begins, by enabling these warnings to the maximum practical degree and by establishing rules to *prevent* disabling them without performing a broader risk analysis. Once implementation begins, developers are likely to repeatedly use constructs that would trigger warnings if they are not told of them immediately, making it potentially more expensive to add warning flags later. A developer may disable warning flags without understanding the long-term risks; while disabling flags is sometimes necessary,

doing so should be considered carefully. The *Software Assurance in Acquisition and Contracting Language* guide specifically points out this technique, saying, "Of the software that are being delivered, were some compiler warning flags disabled? If so, who made the risk decision for disabling it?" [DHS Acq 2012].

### d. Assessment

Pros:

- It is often already available; in such cases, no costs are associated with acquiring and integrating a new tool into the development environment.

- Plans to use warning flags encourage good coding practices from the start of code development.

- It is easily enabled before code development begins.

Cons:

- Warning flags are limited in the problems they can find, in part because the implementers of warning flags must avoid significantly slowing down the compilation process and system operation.

- It can lead to false positives that result in extra work (e.g., to change the code to stop the warning from triggering).

- It can be expensive to enable once the software has been developed. Often such code is full of constructs that trigger warning flags, requiring time-consuming analysis and code changes.

### e. Resource Requirements

This tool/technique is relatively low cost when enabled early, because it is already built into the selected development tools. Resource requirements include:

Resource requirements include:

- Initial resource requirements:
  - Usage effort:
    - To ensure appropriate warning flags are enabled.
    - To modify code in response to raised flags, or documenting and tracking cases where the code continues to trigger warnings.
  - Training: Developer training may be needed, particularly to make it clear that disabling warning flags should not be done lightly. Since warning flags are built into the tools developers already use, the cost of the training tends to be small.

C-5

- Tools and associated computing resources: It may be necessary to move to the latest version of the compiler being used; later versions typically include more warning flags with more refined implementations.

- Recurring resource requirements:

  - Usage effort:

    o To determine whether a raised flag is inappropriate (e.g., its false positive rate is too high) and the flag needs to be disabled.

    o To modify code in response to raised flags, or documenting and tracking cases where the code continues to trigger warnings. This can be a significant effort. It is sometimes possible to slowly add flags and modify code or only apply new flags to new code.

This tool/technique can be much more expensive to apply later in existing code, depending on the code and warning flag involved. The list of resource requirements is the same as above, but often "modifying code to respond to warning flags" can be a significant effort. It is sometimes possible to slowly add flags and modify code or only apply new flags to new code.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name / Flag name | Languages Supported | URL |
|---|---|---|---|
| GNU | gcc options -Wall, -Wextra, -ftrapv | C, C++, Objective-C, Java, Ada | http://gcc.gnu.org/ |
| Microsoft | Visual C++ /W4 | C (89 only), C++, and C++/CLI | http://msdn.microsoft.com/en-us/vstudio//default.aspx |
| Perl | perl -w, use warnings "all" | Perl | http://www.perl.org |

## 3. Source Code Quality Analyzer

### a. Overview

Source code quality analyzers are tools that examine software source code and search for the implementation of poor coding or certain poor architecture practices, using pattern matches against good coding practices or mistakes that can lead to poor

functionality, poor performance, costly maintenance, or security weaknesses depending on context.

### b. Details

Source code quality analyzers typically process the source code into some internal intermediate representation, similar to how a compiler works. They then use rules (patterns) of good coding practices to search this intermediate representation and report matches. The intermediate representation and rule definition approach may use standards or be specific to the tool. Some of these tools have expanded to provide some architectural characteristics such as function point analysis and other mechanisms as a means of identifying where there may be opportunities for improving the quality of software construction as well as reducing the cost of maintenance. In some cases, users can define their own rules.

Most tools use approximations of internal constructs so that they can scale up to cover analysis of large software systems.

These tools can be used in different ways; we have grouped these uses into:

- Spot check – perform a quick quality check of "top" code quality issues, primarily as an "audit" to get a sense of how much risk using the software entails.

- Traditional use – search for code quality issues using the rule sets provided with the tool to identify specific quality issues and guidance on how to fix them. Typically this approach is used when the goal is to improve the quality of the code.

- Context-configured – as with the traditional use, but rule sets are specially created and tailored for evaluating that particular system in its intended environment, and information is added about the system context (e.g., which inputs are untrusted).

Note that the boundary between quality analysis and vulnerability-finding is not sharp. The distinction is in the emphasis of the rule sets. Some tool vendors who started out as code quality analyzer vendors are migrating to also provide source code weakness analysis as well; those are grouped into the category of "Source Code Security Analyzers" (for example, see SAMATE[4]) and "static application security testing" (SAST) tools.

---

[4] http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

### c. Applicability

These tools require source code; typically the source code must compile, and in many cases, it must be possible to build executables. The tool must be able to process the programming language(s) used, and the tool's rule sets must be tailored for that language and environment as well.

Theoretically these tools can apply to any kind of software. In practice, rule sets are typically developed with specific kinds of software in mind (e.g., web server, embedded software, or operating system kernel). Many applications build on frameworks; to be effective, a tool needs to have rule sets designed to work with those frameworks.

### d. Assessment

Pros:

- These tools can quickly find many code quality issues. Some can also provide a discrete set of architectural patterns that can be used to better understand and address code complexity and maintenance concerns.

- They scale better than manual evaluation, since they can be configured to evaluate large code bases and delve into software in depth without getting tired or bored.

Cons:

- Code quality issues that do not match rule set patterns are not detected.

- Most tools are subject to false negatives, that is, they can fail to find poor code quality issues, even if the code quality pattern is in the rule set.

- Most tools are "unsound," that is, they approximate some values during analysis to enable scaling up to large software systems. These approximations can be not-quite-correct, leading to a masking of quality issues in complex systems.

- Many tools produce many false positives. Also, different code quality characteristics will have different levels of assurance impacts, and this relationship cannot be fully determined by a tool. Thus, results require human review that relies on knowledge of the application, language, operational environment, mission, and types of good coding practices.

- Training is necessary. The tools typically report locations and what the potential code quality type is, and perhaps some guidance on mitigation. Turning this into actionable information requires human review to determine whether a reported problem is a false positive, what it means, and what (if anything) should be done.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    - Licensing fees. Licensing schemes vary, some per node, some by lines of code analyzed, and some are open source.

    - Training, including costs for training and the time of those receiving the training. Using these tools requires knowledge and understanding of the tool results, including how to filter out false positives and configure the tool to quickly produce expected results, both as an on-demand or inline operation in the software development lifecycle.

- Recurring resource requirements:

    - Annual maintenance fees.

    - Lines of code fees (for tools that charge per line).

    - Ongoing training of analysis for managing through new configurations, new code quality concerns, identifying new types of false positives, learning new language based code quality concerns.

    - Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

### f. Examples of Suppliers/Products

Various documents discuss or review such tools, e.g., [Emanuelsson 2008] and [Rebel].

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| Synopsys Coverity | Coverity Quality Advisor Coverity Save | C, C++, Java; C# in progress for advisor | http://www.synopsys.com/software/coverity/Pages/default.aspx |
| Klocwork | Truepath® Refactoring® | Truepath: C/C++, Java and C# Refactoring: C/C++ | http://www.klocwork.com/products/insight/index.php |

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| Sonar | SonarSource | C, C++, Java, C#, VB.net, PL/1, COBOL, PHP, Python, VB6, Natural, Javascript, XML, etc; (caveat: when leveraging their open source orchestration engine, only some languages are open source and licensed as such)  Commercial license support includes a broader set of languages. | http://www.sonarsource.org/ |
| CAST | Application Intelligence Platform | Includes .NET, Java, COBOL | http://www.castsoftware.com/products/the-application-intelligence-platform |

## 4. Source Code Weakness Analyzer

### a. Overview

Source code weakness analyzers are tools that examine software source code and search for vulnerabilities, using pattern matches against well-known types of common vulnerabilities (weaknesses).  They are also called "Source Code Security Analyzers" (per SAMATE[5]) and "static application security testing" (SAST) tools.

### b. Details

Similar to source code quality analyzers, source code weakness analyzers typically process the source code into some internal intermediate representation (note that this is similar to how a compiler works).  They then use rules (patterns) to search this intermediate representation for vulnerabilities and report matches.  The intermediate representation and rule definition approach may use standards or be specific to the tool. Examples are patterns of method/function use or "tainted" data from untrusted users that can directly flow to sensitive operations.  In some cases, users can define their own rules.

[Kupsch] reports that in their analysis these types of tools (Fortify and Coverity in their case) found significantly fewer problems than manual review.  On the other hand, manual review is time-consuming and can be difficult to scale up cost-effectively.  "The tools are not perfect, but they do provide value over a human for certain implementation

---

[5] http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

bugs or defects such as resource leaks. They still require a skilled operator to determine the correctness of the results, how to fix the problem and how to make the tool work better."

Note that the boundary between quality analysis and vulnerability-finding is not sharp. The distinction is in the emphasis of the rule sets.

Most tools in practice use approximations of internal constructs so that they can scale up to cover analysis of large software systems.

These tools can be used in different ways; we have grouped these uses into:

- Spot check – perform a quick quality check of "top" weaknesses, primarily as an "audit" to get a sense of how much risk using the software entails.

- Traditional use – search for weaknesses using the rule sets provided with the tool to identify specific vulnerabilities and guidance on how to fix them. Typically this approach is used when the goal is to fix vulnerabilities in the code.

- Context-configured – as with the traditional use, but rule sets are specially created and tailored for evaluating that particular system in its intended environment, and information is added about the system context (e.g., which inputs are untrusted).

### c. Applicability

These tools require source code; typically the source code must compile, and in many cases, it must be possible to build executables. Thus, source code must exist and the tool user must have access to it. The tool must be able to process the language used in writing the software, and its rule sets must be tailored for that language and environment as well.

Theoretically these tools can apply to any kind of software, but rule sets are typically developed with applications in mind (e.g., these tools are less likely to be useful in examining operating system kernels without many additional specialized rules). Many applications build on frameworks; to be effective, a tool needs to have rule sets designed to work with those frameworks.

### d. Assessment

Pros:

- These tools are helpful in quickly finding some common types of vulnerabilities.

- They scale better than human evaluation, since they can manage to evaluate large code bases and delve into software in depth without getting tired or bored.

Cons:

- Some types of vulnerabilities are notoriously difficult to describe using rule sets (e.g., logic bombs); only vulnerabilities that can be described as patterns can be found.

- Vulnerabilities that do not match rule set patterns are not detected.

- Most tools can fail to find vulnerabilities, even if the vulnerability pattern is in the rule set (a.k.a. "false negatives"). Most tools are "unsound," that is, they approximate some values during analysis (the "*invariants*") to enable scaling up to large software systems. These approximations can be not-quite-correct, leading to a masking of vulnerabilities in complex systems.

- Any one tool tends to find a very small percentage of vulnerabilities in an application, even for just the types of vulnerabilities the tool is designed to find. [CAS 2011]

- Many tools produce significant false positives. Also, different vulnerabilities will have different levels of importance that cannot be fully determined by a tool. Thus, results require human review; this review requires knowledge of the application, language, operational environment, mission, and common weakness types.

- Training is necessary. The tools typically report locations and what the potential vulnerability is, and perhaps some guidance on mitigation. Turning this into actionable information requires human review to determine whether it's a false positive, what it means, and what (if anything) should be done.

## e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  - Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

  - Training, including costs for training, and the time of those receiving the training. Using these tools requires knowledge and understanding the tool results, including how to filter out false positives and configure the tool to quickly produce expected results.

- Recurring resource requirements:

  - Annual maintenance fees.

  - Lines of code fees (for tools that charge per line).

− Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

### f. Examples of Suppliers/Products

Longer lists of suppliers/products are available at:

- NIST SAMATE "source code Security analyzers" page[6],

- NSA Center for Assured Software (CAS) evaluations [CAS 2011].

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| Brakeman project | Brakeman | Ruby on Rails | http://brakemanscanner.org/ |
| FindBugs project | FindBugs | Java | http://findbugs.sourceforge.net/ |
| Parasoft | Jtest<br>C++test<br>dotTEST | Java<br>C, C++<br>.NET | https://www.parasoft.com/product/jtest/<br>https://www.parasoft.com/product/cpptest/<br>https://www.parasoft.com/product/dottest/ |
| LLVM Project | Clang Static Analyzer | C, C++, and Objective-C | http://clang-analyzer.llvm.org/ |
| PMD project | PMD | Java, JavaScript, PLSQL, Apache Velocity, XML, XSL | https://pmd.github.io/ |
| Synopsys Coverity | Coverity Static Code Analysis | C, C++, C#, Java, JSP, JavaScript, PHP, Python, ASP .NET, Objective-C | http://www.synopsys.com/software/coverity/Pages/default.aspx |
| Grammatech | CodeSonar | C, C++ | http://www.grammatech.com/products/codesonar/overview.html |
| HP Fortify | Static Code Analyzer (SCA) | C, C++, C# and other .NET languages, COBOL, Java, JavaScript/ AJAX, PHP, PL/SQL, Python, T-SQL | http://www8.hp.com/us/en/software-solutions/software.html?compURl=1338812#.UXBF-MrX-NM |

---

[6] http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| Checkmarx | Checkmarx | Java, C#/.NET, PHP, C, C++, Visual Basic 6.0, VB.NET, Flash, APEX, Ruby, JavaScript, ASP, Android, Objective C, Perl | http://www.checkmarx.com/ |
| IBM (formerly Ounce Labs) | AppScan Source | C, C++, Java, VB.NET, C# | http://www-01.ibm.com/software/awdtools/appscan/ |

## 5. Source Code Knowledge Extractor

### a. Overview

Source code knowledge extractors extract information such as the architecture and design from the source code to aid analysis.

### b. Details

Knowledge extractors extract key information from source code, e.g., to display summary information about the software, enable searches for key information, or to enable access to data managed by the source code.

Knowledge extractors can be used in many other ways; in particular, a knowledge extractor can be used as the technical baseline for implementing a source code quality analyzer or a source code weakness analyzer. This category focuses on using extractors to obtain architectural, design, and mission layer information. If a knowledge extractor is used to implement a different tool/technology type, consult that other category instead. .

These tools can be used in different ways; we have grouped these uses into:

- Traditional use – examine software, e.g., to see its design,

- Context-configured – as with the traditional use, but rule sets are specially created and tailored for evaluating that particular system in its intended environment, and information is added about the system context.

### c. Applicability

These tools require source code; typically the source code must compile, and in many cases, it must be possible to build executables. Thus, source code must exist and the tool user must have access to it. The tool must be able to process the language used in writing the software, and its rule sets must be tailored for that language and environment.

**d. Assessment**

Pros:

- These tools are helpful in dealing with larger codebases (e.g., more than 1 million lines of code). They can be especially helpful when examining codebases new to the analyst.

Cons:

- They do not directly find vulnerabilities, but instead, extract and present information to aid analysts' understanding. Of course, once a design is better understood (e.g., where inputs and outputs occur), it is easier to search for certain kinds of vulnerabilities.

- Training is necessary. Turning this into actionable information requires human review.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

  − Training, including costs for training and the time of those receiving the training. Using these tools requires some knowledge and understanding of the tool results, including how to filter out false positives and configure the tool to quickly produce expected results.

- Recurring resource requirements:

  − Annual maintenance fees,

  − Lines of code fees (for tools that charge per line),

  − Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| IBM | Rational Asset Analyst | http://www-03.ibm.com/software/products/us/en/raa/ |
| Micro Focus | Enterprise Analyzer | http://www.microfocus.com/products/enterprise-analyzer/enterprise-analyzer/index.aspx |

## 6. Traditional Virus/Spyware Scanner

### a. Overview

Traditional virus/spyware scanners are tools that search for known malicious patterns in the binary or bytecode.

Note that modern "anti-virus" programs also perform behavioral analysis; this capability is (for our purposes) rolled into intrusion detection systems (IDSs)/intrusion prevention systems (IPSs), discussed in a separate category.

### b. Details

Traditional virus and spyware scanners are host-based tools that detect, prevent, and remove malware (of various types), including computer viruses, worms, Trojan horses, backdoors, key loggers, rootkits, adware, and spyware. They are also called "anti-virus" and/or "anti-spyware" tools. Traditional anti-virus detection methods include signature-based detection as well as heuristics-based detection:

- Signature based detection is a method of detecting and locating known viruses and malware. In this case, the antivirus software does a comprehensive comparison of the contents of files to a dictionary of virus signatures to identify whether each file is clear of known viruses.

- Heuristic-based detection is a method of detecting and locating unknown threat, e.g., for variants of known malware or for typical damage done. This sort of analysis takes computing resources and time, which slows down system performance. Heuristic detection also increases the number false positives, creating operational or system inefficiencies.

Most traditional virus and spyware scanners in practice are standard solutions used on hosts (desktop, workstation, server, etc.).

### c. Applicability

These tools reside on hosts, including desktops, workstations, and servers.

**d.   Assessment**

Pros:

- These tools are helpful in quickly finding whether a file is corrupt or infected with known malware, viruses, worms, etc.

- These tools are commoditized and thus on a per node licensing model are relatively inexpensive.

- There are multiple vendors with extensive research capabilities, providing constant updates to newly identified signatures and/or developed heuristics for continuous improvement in the quality of analysis.

- There is extensive community participation and communication for alerts on new viruses, techniques, and heuristics.

Cons:

- Constant updates to signatures and heuristics are necessary.  Managing the scale of these updates across an enterprise is a challenge and creates a significant expenditure.

- False positives can occur, especially when applying heuristics.

**e.   Resource Requirements**

Resource requirements include:

- Initial resource requirements:

    - License for software to run and managing default configurations to ensure consistency across the enterprise.

    - Minimal training requirement for implementing traditional anti-virus or anti-spyware tools.

- Recurring resource requirements:

    - Annual licensing model,

    - Ensuring the most current updates are tested and deployed across the enterprise to deliver maximum protection.  This may include balancing this with the DoD Information Assurance Vulnerability Alert (IAVA) process.

**f.   Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Description | URL |
|---|---|---|---|
| McAfee | McAfee AntiVirus Plus | Anti-virus | http://www.mcafee.com/us/ |
| Symantec | Norton Anti-virus | Anti-virus | http://www.symantec.com/index.jsp |
| Trend Micro | Titanium Antivirus Plus | Anti-virus | http://www.trendmicro.com/us/indexnight.html |

## 7. Bytecode Weakness Analyzer

### a. Overview

Bytecode weakness analyzers are tools that examine bytecode and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses). These types of tools are also called bytecode scanners (http://samate.nist.gov/index.php/Byte_Code_Scanners.html). These are in concept similar to source code weakness analyzers and binary weakness analyzers, but for bytecode instead of source code or binary (executables) respectively.

### b. Details

Many implementations, such as common Java implementations, generate an intermediate "bytecode" as an executable. Bytecode weakness analyzers typically process the bytecode into some internal intermediate representation. They then use rules (patterns) of common weaknesses to search this intermediate representation, and report matches. The intermediate representation and rule definition approach may use standards or be specific to the tool. In some cases, users can define their own rules.

Most tools in practice use approximations of internal constructs so that they can scale up to cover analysis of large software systems.

Bytecodes tend to be at a higher level of abstraction than traditional binary executables, and they often include useful information such as symbol tables (that list names assigned to various constructs). Thus, it tends to be easier to develop bytecode analyzers than traditional binary tools, and this ease can translate into stronger analysis by tools. However, some information available to source code analyzers is not available to bytecode analyzers.

### c. Applicability

These tools require bytecode, thus, they only apply to language environments that generate bytecode (e.g., many Java implementations).

If bytecode is generated, it is more readily available than source code from proprietary third-party suppliers.

As with source code and binary analyzers, many applications build on frameworks; to be effective, the tool needs to have rule sets designed to work with those frameworks.

### d. Assessment

The pros and cons are similar to those of source code weakness analyzers.

Pros:

- These tools are helpful in quickly finding some common types of vulnerabilities.

- They scale better than human evaluation, since they can manage to evaluate large code bases and delve into software in depth without getting tired or bored.

- Bytecode is more readily available from proprietary COTS suppliers, if there is bytecode at all.

Cons:

- Some types of vulnerabilities are notoriously difficult to describe using rule sets (e.g., logic bombs); only vulnerabilities that can be described as patterns can be found.

- Vulnerabilities that do not match rule set patterns are not detected.

- Most tools can fail to find vulnerabilities, even if the vulnerability pattern is in the rule set (a.k.a. "false negatives"). Most tools are "unsound," that is, they approximate some values during analysis (the "*invariants*") to enable scaling up to large software systems. These approximations can be not-quite-correct, leading to a masking of vulnerabilities in complex systems.

- Any one tool tends to find a very small percentage of vulnerabilities in an application, even for just the types of vulnerabilities the tool is designed to find. [CAS 2012]

- Many tools produce a significant number of false positives. Also, different vulnerabilities will have different levels of importance that cannot be fully determined by a tool. Thus, results require human review; this review requires knowledge of the application, language, operational environment, mission, and common weakness types.

- Due to the lower-level nature of the tools, it can be very difficult for external parties to determine whether a report is a false positive or not.

- Even if a report is confirmed to be a problem, an external party would typically have difficulty fixing the problem, and any such attempt would void any support or warranty.

- Training is necessary. The tools typically report locations and what the potential vulnerability is, and perhaps issue some guidance on mitigation. Turning this into actionable information requires human review to determine whether it's a false positive, what it means, and what (if anything) should be done.

For detailed evaluations of tools, refer to [CAS 2012].

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Licensing fees. Licensing schemes vary, some per node, others by lines of code analyzed.

  − Training, including costs for training, and the time of those receiving the training. Practically using these tools requires knowledge and understanding of the tool results, including how to filter out false positives and configure the tool to quickly produce expected results.

- Recurring resource requirements:

  − Annual maintenance fees,

  − Lines of code fees (for tools that charge per line),

  − Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority. For bytecode analyzers, this can be significant, due to the lower-level nature of the data they analyze.

### f. Examples of Suppliers/Products

Longer lists of suppliers/products are available at the NIST SAMATE "source code Security analyzers" page, http://samate.nist.gov/index.php/Byte_Code_Scanners.html.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Bytecode Supported | URL |
|---|---|---|---|
| FindBugs project | FindBugs | Java | http://findbugs.sourceforge.net/ |
| FindSecurityBugs project | FindSecurityBugs (plug-in to FindBugs) | Java | http://h3xstream.github.io/find-sec-bugs/ |
| Mono Project | Gendarme | .NET | http://www.mono-project.com/Gendarme |
| VeraCode | VeraCode Static Analysis | Java, .NET | http://www.veracode.com/ |

## 8. Binary Weakness Analyzer

### a. Overview

Binary weakness analyzers are tools that examine binaries (executables) and search for vulnerabilities, using pattern matches against well-known common types of vulnerabilities (weaknesses).  They are also called binary code scanners (http://samate.nist.gov/index.php/Binary_Code_Scanners.html).  They are in concept similar to source code weakness analyzers and bytecode weakness analyzers, but for binary executables bytecode instead of source code or bytecode respectively.

### b. Details

Binary code weakness analyzers typically process the binary code into some internal intermediate representation.  They then use rules (patterns) of common weaknesses to search this intermediate representation, and report matches.  The intermediate representation and rule definition approach may use standards or be specific to the tool. In some cases, users can define their own rules.

Most tools in practice use approximations of internal constructs so that they can scale up to cover analysis of large software systems.

Binary executables are at a lower level of abstraction than typical bytecodes, and are at a far lower level of abstraction than source code.  Thus, it tends to be more difficult to develop binary analyzers compared to bytecode or source analyzers.  This difficulty can translate into difficulties for analyzer developers; significant algorithms and computation may be required to determine what is obvious from the source code or, if available, a bytecode.  This difficulty can translate into greater difficulty in finding weakness patterns.

Some tools need some help from the developers of the executable; e.g., they may require the development organization to provide a "symbol table."  In such cases, some

cooperation from the developer is required. On the other hand, symbol tables may be significantly easier to get from a developer than the source code. This need for additional information affects when the tool can be used for the assessment of proprietary COTS software.

### c. Applicability

These tools require binaries, thus, they only apply when binaries are available. Many language implementations typically generate binaries (including C, C++, and Objective-C). Many providers of proprietary COTS software will provide binaries even if they do not provide source code.

Many applications build on frameworks; to be effective, a tool needs to have rule sets designed to work with those frameworks.

### d. Assessment

The pros and cons are similar to source code and bytecode weakness analyzers.

Pros:

- These tools are helpful in quickly finding some common types of vulnerabilities.

- They scale better than human evaluation, since they can manage to evaluate large code bases and delve into software in depth without getting tired or bored.

- Binaries are more readily available from proprietary COTS suppliers, if there is a binary at all.

Cons:

- Some types of vulnerabilities are notoriously difficult to describe using rule sets (e.g., logic bombs); only vulnerabilities that can be described as patterns can be found.

- Vulnerabilities that do not match rule set patterns are not detected.

- Most tools can fail to find vulnerabilities, even if the vulnerability pattern is in the rule set (a.k.a. "false negatives"). Most tools are "unsound," that is, they approximate some values during analysis (the "*invariants*") to enable scaling up to large software systems. These approximations can be not-quite-correct, leading to a masking of vulnerabilities in complex systems.

- Any one tool tends to find a very small percentage of vulnerabilities in an application, even for the types of vulnerabilities the tool is designed to find [CAS 2012].

- Many tools produce a significant number of false positives.  Also, different vulnerabilities will have different levels of importance that cannot be fully determined by a tool.  Thus, results require human review; this review requires knowledge of the application, language, operational environment, mission, and common weakness types.

- Due to the low-level nature of the tools, it can be very difficult for external parties to determine whether a report is a false positive or not.

- Even if a report is confirmed to be a problem, an external party would typically have difficulty fixing the problem, and any such attempt would void any support or warranty.

- Training is necessary.  The tools typically report locations and what the potential vulnerability is, and perhaps some guidance on mitigation.  Turning this into actionable information requires human review to determine whether it's a false positive, what it means, and what (if anything) should be done.

For detailed evaluations of tools, refer to [CAS 2012].

e.  **Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  − Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.

  − Training, including costs for training, and the time of those receiving the training.  Practically using these tools requires knowledge and understanding of the tool results, including how to filter out false positives and configure the tool to quickly produce expected results.

- Recurring resource requirements:

  − Annual maintenance fees,

  − Lines of code fees (for tools that charge per line),

  − Usage effort.  This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.  For binary tools these are quite substantial, due to the low-level nature of the data they analyze.

### f. Examples of Suppliers/Products

Longer lists of suppliers/products are available at the NIST SAMATE "binary code scanners" page.[7]

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| VeraCode | VeraCode Static Analysis | http://www.veracode.com |
| GrammaTech | CodeSonar for Binaries (builds on CodeSurfer/x86) | http://www.grammatech.com/codesonar |

## 9. Inter-application Flow Analyzer

### a. Overview

Inter-application flow analyzers are tools that examine the control and/or data flows of a set of applications, identifying their communication interfaces (such as Android intents) and permissions, and then identify flows that violate the security policy.

### b. Details

Although inter-software program or application flow analyzers have existed in enterprise systems analysis for many years, the use of such an analysis in the mobile environment is new. Mobile application inter-application flow analysis can be done by extracting information from application source, bytecode, or binary. Mobile applications are built on top of a large pre-existing framework, making it somewhat more manageable to develop tools to analyze inter-software application flows. Mobile environments typically do isolate individual applications. However, by default they do not counter collusion or the unexpected exploitation of one application's services by another.

In the case of an enterprise mobile environment, this type of tool enables verification that the applications accepted into the application store comply with security policy for inter-application communication. Static analysis may identify all possible pathways of communications among the mobile applications (in practice, it can sometimes be difficult to truly identify all possible pathways, but dynamic approaches certainly cannot be certain to identify all possible pathways). These types of tools extract information from the application package manifest (where available), use automated

---

[7] http://samate.nist.gov/index.php/Binary_Code_Scanners.html

static analysis techniques to extract up the interdependencies on the application code, derive possible information flows from application sources and sinks leveraging control flows from application entry points to show all possible outcomes.

For example, app components are the fundamental building blocks of Android applications. Application components can be activities (which provide a user interface to an application), services (which perform an action in the background, broadcast receivers (which listen for messages from other applications), or content providers (which store potentially shareable data, and component communications using intents). "Because the system runs each app in a separate process with file permissions that restrict access to other apps, [an] app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component [if authorized]." [Android Fundamentals 2013]

### c. Applicability

This type of analysis applies any time there are multiple interacting applications that may together collude or cause vulnerabilities. This seems especially appropriate in the mobile environment, where enterprise data and applications that should not have access to enterprise data might be on the same device. This type of tool can be used to analyze sets of applications in an application store, which could lead to results that allow some sets of applications to be acceptable as long as certain other applications are not also installed on the same device. In particular, the role individuals or groups of individuals play in an environment can define the baseline set of applications necessary for operations. A baseline set can be examined to determine risk, and additional applications can also be examined to determine changes to the risk profile if they are added to a baseline.

### d. Assessment

Pros:

- Targeted approach to quickly ensure application collusion risks are understood.

- Can use for assessing third-party applications without requesting sources.

Cons:

- Current implementations support only Android mobile applications.

- In some cases it can be difficult to determine this information using solely static analysis.

### e. Resource Requirements

Resource requirements include initial licenses and annual maintenance fees.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Galois | FUSE[8] | Android | http://corp.galois.com |
| - | Lintent | Android | http://www.dais.unive.it/~calzavara/papers/forte13.pdf |

## 10. Binary/Bytecode Simple Extractor

### a. Overview

Binary/bytecode simple extractors are simple tools that report simple facts about binary executables or bytecode, or perform trivial analysis of them.

### b. Details

Binary and bytecode file formats[9] are not easy to read directly. There are various simple tools that can report simple facts, or perform trivial analysis, of binary executables or bytecode. Since binaries and bytecode by their nature are opaque, these tools can provide some quick – but extremely limited – insight.

A useful example is the Portable Operating System Interface (POSIX) "strings" command, which writes out any sequence of four or more printable characters terminated by the newline or NUL[10] character [POSIX.1-2008]. Other examples include GNU readelf, which reports low-level details about executable files if they are in Executable and Linkable Format (ELF) format. Other tools exist for a variety of formats.

---

[8] DARPA funded initial development of FUSE for Android; it is currently available for use.

[9] Binary formats include the ELF format used by most Unix and Linux systems, as well as the Microsoft Portable Executable (PE) and Common Object File Format (COFF) files used for executables on the Microsoft Windows operating systems. One bytecode format is the JAR (Java ARchive) format, which in turn can include class files for the Java Virtual Machine (JVM). Bytecode for the Microsoft .NET environment can be in PE format, which contains Common Language Infrastructure (CLI) assemblies that house Common Intermediate Language (CIL) code.

[10] NUL is the conventional name of the null character; in ASCII this is encoded as the number 0.

### c. Applicability

These tools only apply when a bytecode or binary is available.  They could be used even when source code is available, but since they only provide very limited insight, they have even less value when source code is available.

By themselves, these tools are extremely limited in the information they can provide.  In general they can only provide superficial, basic information about a bytecode or binary.  In rare cases, such as an incompetent malicious developer who inserts malicious code into a bytecode or binary, these tools can reveal problems such as trapdoors (by revealing an unexpected string in an executable that turns out to be a trapdoor initiator).  They can also reveal unintentional trapdoors left by developers who inserted them for debugging or testing.  By themselves, these tools are used primarily because they have typically no or little cost and many developers already know how to use them.

As a first step towards a deeper analysis, these tools have more merit, because they can quickly provide some basic information about the TOE before bringing more powerful tools to bear.

### d. Assessment

Pros:

- These tools are helpful in quickly reporting basic information about binaries or bytecodes.

- They are cheap/free and often already available to developers.

- They can be helpful before bringing more powerful tools to bear.

Cons:

- By themselves they usually find nothing actionable.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Training, including costs for training, and the time of those receiving the training.  Typically little for developers.

    - Recurring resource requirements:
    - Usage effort.  This is primarily for reviewing results, e.g., to determine whether any "strings" or binary segments are unexpected.  Often this is small, but often the results are not conclusive.

**f. Examples of Suppliers/Products**

These tools are typically included as part of an operating system or development environment.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name |
|----------|--------------|
| GNU | strings |
| GNU | readelf |

# 11. Compare Binary/Bytecode to Application Permission Manifest

**a. Overview**

Tools that compare binary/bytecode to the application permission manifest examine the binary/bytecode to determine what permissions the application attempts to use and compare that to the permissions actually requested in the application permission manifest. Note that "permissions" in this context are the privileges granted to an application, not the permissions set on objects such as files or memory.

**b. Details**

Such tools must estimate the expected permission requests by analyzing the binary/bytecode. Some technical constructs (such as reflection) can make it difficult to determine exactly what permissions are required.

**c. Applicability**

These tools can potentially identify under-granting or over-granting of privileges to a given application.

**d. Assessment**

Pros:

- Simple and easy to apply.

Cons:

- They do not directly find vulnerabilities, but instead warn of potential inconsistencies between claimed privilege requirements and actual requirements.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Training, including costs for training, and the time of those receiving the training. Typically little for developers.

- Recurring resource requirements:

  − Usage effort. This is primarily for reviewing results.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Stowaway | Stowaway | http://www.cs.berkeley.edu/~afelt/Android_permissions.pdf |
| ViaForensics | ViaLab (portion) | https://viaforensics.com/products/vialab/ |

Note that current evidence suggests that Stowaway is no longer maintained.

## 12. Obfuscated Code Detection

### a. Overview

Obfuscated code detectors detect when code is rendered obscure. They may be applied to source code (e.g., JavaScript), bytecode, or executables. Obfuscation is a commonly used technique for protecting critical or proprietary technology, so that others cannot easily determine what the software does. However, obfuscation can make it more difficult to identify unintentional and intentional vulnerabilities. In particular, obfuscation can be used to obscure malware. Obfuscated code detectors are a way to counter the risk of obscured vulnerabilities.

### b. Details

Obfuscation is used to protect critical or proprietary technology. Obfuscation approaches can limit or counter reverse-engineering methods used to better understand software. Note that some DoD projects may be required to release obfuscated code, and many proprietary software application developers use obfuscation to protect their intellectual rights. However, obfuscation is also a commonly used practice for disguising

malicious code. Since attackers also leverage obfuscation, the challenge is that obfuscation can limit the ability of other analyzers to identify indicators of risk.

In order to reduce the risk of obfuscated code that may or may not have both unintentional vulnerabilities and intentional (malware), tools have been developed to detect when code is rendered obscure. These tools can detect whether code is obfuscated, providing relevant information for risk decisions.

Code can be obfuscated in a variety of ways. Some practices can obfuscate code in a limited manner, even if that is not their primarily purpose. Such practices include generating executables (instead of distributing source code) and using mechanisms that cause are more difficult to statically analyze (e.g., reflection). However, these general practices can often be addressed by designing tools to deal with them. In this type of tool/technique we focus on mechanisms that have, as a primary purpose, the goal of limiting or countering reverse-engineering. For our purposes this can include source code minification.

### c. Applicability

Obfuscated code detection is a fairly simple activity when using tools that detect whether code is obfuscated or not. This can help provide a quick indicator in how effective many other analyzers are likely to be. If a program is obfuscated, then other tools (especially static analysis tools) may be ineffective, suggesting that the obfuscated program has a higher risk unless other steps can be taken.

### d. Assessment

Pros:

- Fast ramp-up for use,
- Fast method of identifying obfuscation,
- Cost effective approach.

Cons:

- Limited knowledge for assessment.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Small investment for licensing

- Recurring resource requirements:
  - Annual maintenance fees are often minimal to none.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| ViaForensics | Vialab (portion) | Various | https://viaforensics.com/products /vialab/ |

# 13. Binary/Bytecode Disassembler

**a. Overview**

Binary/bytecode disassemblers recover higher-level constructs from lower-level binaries and bytecode, which can then be analyzed by people.

**b. Details**

Since binary and bytecode files are difficult for humans to review directly, and for many tools to analyze, one approach is to use automated tools to reconstruct a representation of the executable that is easier to review and analyze. These tools are often called "disassemblers" if they produce more-or-less direct representations of the underlying binary or bytecode, and "decompilers" if they produce higher-level source code representations for a higher-level language, though this distinction is not always made consistently.

In theory, this approach could find all vulnerabilities, unintentional or intentional, since what is being reviewed is what is executed. However, the large effort and strong expertise required to do this limits in practice what this approach can and cannot do with large, modern software.

**c. Applicability**

These tools require binary or bytecode. They often are not be used if source code is available, although they could be if there was doubt that the binary or bytecode faithfully represented the source code.

**d. Assessment**

Pros:

- They can apply in cases where only the binary or bytecode is available.

Cons:

- Human analysis of disassembled/decompiled results tends to be very costly and difficult to scale.

- Supporting automated toolsets may have trouble processing disassembled/decompiled results for weaknesses.

- Training is necessary.  Typically users must understand the lower-level constructs that are being analyzed.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:
  - Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.
  - Training, including costs for training, and the time of those receiving the training.  This includes in-depth knowledge of the underlying binary or bytecode format.

- Recurring resource requirements:
  - Annual maintenance fees.
  - Usage effort.  This can be substantial, especially for human (manual) review of results.

**f. Examples of Suppliers/Products**

The NIST SAMATE "binary code scanners" page has more information: http://samate.nist.gov/index.php/Binary_Code_Scanners.html.  Note that IDA Pro is a commercial product name and is not related to the Institute for Defense Analyses (IDA) and derives its identify from "**IDA** is the **I**nteractive **D**is**A**ssembler."  Also, IDA Pro is both a disassembler (static) and a debugger (dynamic).

The following is an example of a supplier and product.  This example is provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|----------|-------------|---------------------|-----|
| Hex-Rays | IDA Pro | | https://www.hex-rays.com/index.shtml |

## 14. Focused Manual Spot Check

### a. Overview

A focused manual spot check is a technique that focuses manual analysis of code (typically less than 100 lines of code) to answer specific questions. For example, does the software require authorization when it should? Do the software interfaces contain input checking and validation?

### b. Details

Manual spot-checking of code can be performed as source, byte or binary static analysis. Analyzing bytecode or binary code may require sophisticated manual reviewers.

Spot-checking of source code can be an efficient approach to enforcing good coding standards, compliance with specific architecture and interface requirements, or checking for typical weaknesses (such as the OWASP top 10 or top 25 CWE/SANS list) that will require correction by developers prior to release of code.

Random spot-checking style of code can provide insight into the software. It can also help measure the level of developer skill/knowledge, as well as potential gaps in education and training of the development team.

### c. Applicability

It can apply to all software, but the costs of manual spot-checking typically limit it to small portions of the software.

It typically used to focus on very specific areas. Examples include authorization functionality and input checking/validation of untrusted input.

### d. Assessment

Pros:

- Depending on the focus of the spot-checking, it can be quite effective to remove some types of weaknesses (e.g., those related to authentication and input validation).

- It can reduce false positives, since humans can use their knowledge of system context.

Cons:

- As the sample size and the system complexity increase, it can become costly, time-consuming, and constrained by the lack of adequately trained people.

- Personnel training and the knowledgebase must be continuously improved to keep pace with context and updates to software.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    - Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

    - Changes to the software may require a continuous monitoring of the code thus making it a recurring activity.

    - Continuous training of developers/analysts to expand knowledge base on new weakness or non-compliance issues.

    - Outsourcing (if applicable of the spot check) to third-party assessor.

### f. Examples of Suppliers/Products

Not applicable.

## 15. Manual Code Review

### a. Overview

Manual code review (other than inspections) is a specialized technique that is the manual examination of code, e.g., to look for malicious code. The manual process can be incorporated into the software development lifecycle either to provide analysis to all code in the software development lifecycle (SDLC).

### b. Details

Manual code review can be used to find coding errors in software. It can be implemented as one of the SDLC process steps with a second person check within the developer organization, or with a third-party evaluator. Depending on the need and focus

of the organization for using manual code review, managing resources for scale is critical. Focusing on specific vulnerability types, critical components requiring more rigor due to their mission critical nature, or a combination of the two, is recommended to manage scale and costs.

Manual source code review can be a complementary technique to help understand software. It can be used for checking coding style to better understand the developer's capabilities, identifying a "top" set of vulnerabilities/weaknesses for corrective action, design validation (e.g., to check the implementation of memory allocation or input validation), configuration, interface requirements validation, review of attack surface, including input/output path analysis, and much more. Manual code review, in cases where the components are critical, can be used as a redundant technique to greatly reduce human error.

### c. Applicability

Since the advantage and disadvantage of manual source code analysis is the dependency on the involvement of a human reviewer, there is a direct correlation between the results yielded and the reviewer's experience with specific technologies, mission/system knowledge, architecture, and various attack/threat scenarios, as well as real-time feedback, including recommendations.

An advantage of manual source code analysis is that humans can be very good at correlation, synthesis, and impact analysis, taking into account a variety of contextual information including system, threat, and vulnerability information.

Code reviewing is more expensive than many other approaches, so it is often used for critical components or processes in a system where other techniques are not able to provide the comprehensive coverage that may be required.

One should be cautious in that manual review does introduce human error, and thus, some errors may not detected. Also, the reviewer is susceptible to fatigue and possibly boredom, creating the potential for inefficiencies and error.

Some of the many articles that this discuss this include [Chmielewski 2013] and [Kesäniemi 2009].

### d. Assessment

Pros:

- Depending on the level of effort and focus of manual source code review, it can be quite effective in detecting specific weaknesses.

- It enables the use of continuous learning, impact analysis, feedback, and recommendation approaches, which are decided advantages of using human review.

- It has a low false positive rate.

Cons:

- Scaling can still be an issue depending on the complexity of the code built.

- Personnel training and knowledgebase must be continuously improved to keep pace with context and updates to software.

- Reviewers are susceptible to fatigue and boredom, reducing its effectiveness.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  − Changes to the software may require continuous monitoring of the code, thus making it a recurring activity and thus a recurring cost.

  − Continuous training of developers/analysts to expand the knowledgebase on new weakness or non-compliance issues.

  − Outsourcing (if applicable of the spot check) to a third-party assessor.

### f. Examples of Suppliers/Products

Not applicable.

## 16. IEEE 1028 Inspections

### a. Overview

An IEEE 1028 inspection is a systematic peer examination to detect and identify software product anomalies.

### b. Details

Inspections, as defined by IEEE Standard 1028, include two to six human participants (including the author) with a rigorous set of roles and processes. The inspection team includes one or more "readers" who lead "the inspection team through

the software product in a comprehensive and logical fashion, interpreting sections of the work (for example, generally paraphrasing groups of one to three lines), and highlighting important aspects. It also includes the "author" who is responsible for contributing to the inspection (based on special understanding) and performing any required rework. [IEEE 2008]

Typical inspection rates are two to three pages per hour for requirements, three to four pages per hour for designs, 100 to 200 lines per hour for source code, and five to seven pages per hour for test plans.

Numerous scientific experiments have shown that inspections can be very effective at finding defects when measured in terms of defects found per hour invested (see [Wheeler 1996] for information on these experiments and other related information).

### c. Applicability

IEEE 1028 inspections can be applied to any software product, including requirements, design, source code, and test plans.

Generally inspections are applied by teams during development, and IEEE 1028 inspections include the author as a required participant. There is little experience in using inspections without participation by an author.

### d. Assessment

Pros:

- IEEE 1028 inspections are effective in finding defects in terms of defects per hour invested (figures vary, but often one defect is found per hour invested).

- Inspections can be applied early in the development lifecycle.

Cons:

- Significant effort and time are needed to perform inspections.

- It can be difficult to get willing and competent participants.

- Inspection requires author participation and detailed knowledge by other participants. Thus, while it is used in some development processes, it is not usually used (or useful) for third-party evaluation (other than to determine whether they have been done).

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

- Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

    - Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

### f. Examples of Suppliers/Products

This is not a "product" in the traditional sense, so listing suppliers is not relevant.

## 17. Generated Code Inspection

### a. Overview

This technique examines generated binary or bytecode to determine that it accurately represents the source code.

### b. Details

This process can detect some cases in which a compiler has "optimized away" and incorrectly implemented important functionality. In particular, it is sometimes important to erase sensitive information (such as unencrypted keys, passwords, and similar sensitive data), but modern compilers will sometimes "optimize away" the erasure code because they detect that the data is not directly used afterwards. Generated code inspection can be used to ensure that the data is actually being erased. If a compiler or later process inserts malicious code, this technique may also detect it.

This process is usually a spot check (e.g., on key or suspicious areas) and not performed across all of the code. However, it *could* be performed across all code if this were considered necessary.

Some compilers have options to simplify this analysis, by generating information to simplify human comparison.

### c. Applicability

These tools require source code and the resulting executable (bytecode or binary code).

**d. Assessment**

Pros:

- The approach can reveal malicious and unintentional vulnerabilities that are difficult to find in many other ways.

Cons:

- It requires in-depth knowledge of the executable format, as well as that of the source language; many developers do not have this knowledge.

- It is expensive if applied in more than a few selected places.

- Later recompilation reduces the credibility of the results, especially if the build environment has changed significantly.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  - Training, including costs for training, and the time of those receiving the training. Practically using these tools requires some knowledge to understand the tool results, including how to filter out false positives and configure the tool to quickly produce expected results.

- Recurring resource requirements:

  - Analysis of each position.

**f. Examples of Suppliers/Products**

This technique typically uses tools already available, e.g., compilers or disassemblers.

## 18. Safer Languages

**a. Overview**

Choosing safer languages is the selection of languages, or language subsets, that eliminate or make it more difficult to inadvertently insert vulnerabilities. This includes the selection of memory-safe and type-safe languages.

**b. Details**

Choosing safer languages including the selection of a language developed to limit/reduce the number of inherent limitations that cause quality and security flaws in the system code.

Computer languages that support mechanisms such as array access outside array boundaries, arbitrary pointer arithmetic, arbitrary type casting, and manual memory management, are generally not considered safe. Examples of unsafe languages include C, C++, and Objective-C. Issues that can arise from a lack of safety include buffer overflow, dynamic memory errors, dangling pointers, and uninitialized variables.

In contrast, languages can be designed to counter common problems. For example, type-safe languages counter some erroneous or undesirable program behavior that can be caused by, for example, a discrepancy between how a data value is initialized versus how it is later used. Type safety can be static (declared in program text) or dynamic (checked at run-time).

Most other languages provide stronger safety characteristics. Languages that are typically considered safe include Java, C#, Ada, and Python. Most languages have mechanisms to temporarily disable safety mechanisms, but these can be strongly localized. The way you evaluate a language varies depending on the extent to which the context dictates safe construction and what performance characteristics the language type can provide to ensure mission/operational requirements are met.

### c. Applicability

Languages are chosen relatively early in the software development lifecycle, as part of design. It is best to identify, during requirements definition, the language requirements to improve the assurance characteristics of the resulting software. This can occur during the first-time build (green-field) of a software system, or during transformation of a legacy system from a "less safe language" (e.g., C, C++, and Objective-C) to a "safer language" (e.g., Java and C#).

### d. Assessment

No language is perfect. Every language has inherent flaws or characteristics that make it either ideal or not ideal as a platform of choice for building or transforming software systems with limited security or quality flaws. Selecting safer languages improves the quality of the implementation and operation of software systems.

Whether addressing a green-field development or existing operational system requiring transformation, limitations such as budgets, resources, knowledge, skills, and timelines, can limit the language type used. Each context must be addressed considering the mission and assurance requirements for the system.

Note that a change in language platform is likely to be dictated by function or mission needs, and not by assurance.

### e. Resource Requirements

- Developers/analysts with knowledge of the language platform chosen,

- Training to identify the best suited language for the context of the software system being built or transformed, and then training in the actual language selected if necessary,

- Tools to support the selected language(s).

### f. Examples of Suppliers/Products

These are essentially the suppliers of various compilers and interpreters.

## 19. Secure Library Selection

### a. Overview

Secure libraries provide mechanisms designed to simplify developing secure applications. They may be stand-alone or be built into larger libraries and platforms.

### b. Details

Secure libraries include mechanisms such as tools to authenticate users, encrypt and manage sessions, validate specific types of common input (e.g., email addresses, URLs, and HTML data), invoke other tools without risking injection attacks, and filter information back.

Developers can, of course, implement such mechanisms themselves, but many developers do not have the necessary background to *correctly* implement them. What's more, by concentrating such functionality into a library used by many developers, any improvements or time spent on the library has the potential to aid many programs.

The divide between "secure libraries" and other kinds of libraries is necessarily porous, since such libraries can be embedded in larger libraries, and traditional libraries are often extended with additional functionality that may include security functionality.

Libraries should be updated as new versions become available and old ones stop being maintained. This is particularly important if a vulnerability is fixed in a library. Pedigree analysis, discussed separately, can identify libraries that are obsolete or vulnerable.

### c. Applicability

Secure libraries must be used to have any positive effect, so they must be made available to developers during design and implementation. Typically this cannot be done with third-party off-the-shelf-components without effort.

**d. Assessment**

Pros:

- Secure libraries concentrate common needs in one place, so that experts can focus on doing it correctly in one place and any corrections will apply to many applications.

- They can reduce development time and effort.

Cons:

- Typically impractical to retrofit on third-party software, and requires effort to retrofit existing software to use them.

- A vulnerability in the library can have wide effects (although once found, its repair also has wide effects).

- Libraries must be kept up-to-date.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  − Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.

  − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  − Annual maintenance fees, depending on the library.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| Apache | Shiro | Java | http://shiro.apache.org/ |
| OWASP | OWASP Enterprise Security API) | Java; many others in various stages of maturity | https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API |

## 20. Secured Operating System Overview

### a. Overview

A secured operating system (OS) is an underlying operating system and platform that is hardened to reduce the number, exploitability, and impact of vulnerabilities.

### b. Details

A secured operating system and platform typically uses a variety of mechanisms to counter adversaries. These include:

- Minimizing the attack surface (the available vectors of attack) by avoiding the installation of unnecessary software and services, removing unnecessary software or services, reducing the number of authorized users, reducing user and program privileges, closing network ports, and so on. Splitting systems into multiple single-function systems often aids this, since each single-function system can have a much smaller set of permitted functions.

- Enabling address space layout randomization (ASLR), which hinders some attacks by making it more difficult for an attacker to predict system addresses. Note that on some platforms executables must be created in a certain way to take advantage, or full advantage, of ASLR, though this is usually not difficult to do if the source code is available.

- Using specially-devised operating systems and platforms with a small verified trusted computing base (TCB) at higher levels of assurance.

Note that this is merely a sample of mechanisms, not a complete list.

### c. Applicability

Applying a secured operating system or platform is broadly applicable to any software. Sometimes a vulnerability built into an application program can be countered, or at least have its impact reduced, through the use of a secure operating system.

However, there are significant limits to what this approach can achieve. While a secured operating system or platform can counter some vulnerabilities, there are many others they cannot counter.

Specially-devised operating systems can be developed as high assurance (HA) systems. These are typically implemented using a small TCB that can be examined in detail. These can provide stronger defenses, but since these often trade away functionality for security, applications must often be specifically designed to work on them. Different products provide varying levels of this trade-off.

**d. Assessment**

Pros:

- It quickly provides some additional protective measures.

Cons:

- It can only provide limited defenses for applications with vulnerabilities.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

    − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

    − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

    − Annual maintenance fees, depending on the library.

**f. Examples of Suppliers/Products**

All operating systems and platforms provide some hardening mechanisms, so there is little point in listing them.

## 21. Origin Analyzer

**a. Overview**

Origin analyzers are tools that analyze source code, bytecode, or binary code to determine their origins (e.g., pedigree and version). From this information, some estimate of riskiness may be determined, including the potential identification of obsolete/vulnerable libraries and reused code.

**b. Details**

Software is typically developed from many sub-components, including embedded libraries. Tools can be used to examine software by comparing pieces of the software with databases of known software components and libraries to determine information such as pedigree and version. This can be done with source code, bytecodes, or binaries; it is typically more difficult to match binaries (even when there are matches).

A common use of such tools is to determine license compliance for open source software. Open source software licenses permit many uses, but typically impose some license requirements depending on the license. By comparing software to a database of open source software, it is sometimes possible to determine the origins and versions, and thus identify licenses not previously disclosed. In theory such license analysis would apply to any software, but it is relatively easy to create a database of publicly released open source software, and much more difficult to create a database of proprietary and custom software. Companies have worked to extend their databases with the latter, but they will typically have far more information available on open source software. However, since these tools can more broadly identify software sub-components, the tools can be used for purposes other than license compliance.

One use is pedigree analysis, that is, determining the original origin of the software and possibly how it ended up in this software. The origin of that software may help risk assessment (e.g., if that origin has known additional risks or is especially reliable).

Another use is to identify obsolete and/or known vulnerable libraries and reused code. A recent study [Williams 2012] provides strong evidence that application developers often do not update the libraries they use. "If people were updating their libraries, [older libraries' popularity would] drop to zero within the first two years. [But popularity extends] over six years. One possible explanation is that some projects, perhaps new development efforts, tend to use the latest version of a library [and then] incremental releases of legacy applications are not being updated to use the latest versions of libraries…." Such tools can identify libraries (including deeply embedded libraries) that have known vulnerabilities. Such vulnerabilities may or may not be exploitable, but the presence of known-vulnerable libraries can serve as a warning for potential problems that can be investigated further.

### c. Applicability

These analyses can be applied to any software, including third-party proprietary software. If the analysis reports significant use of obsolete or known-vulnerable software, it suggests an increased risk in using the software.

### d. Assessment

Pros:

- Origin analyzers can be applied to any software, including third-party proprietary software.

Cons:

- They will not identify libraries not in the database. Thus, these tools are especially likely to identify publicly released open source software, less likely to

identify third-party proprietary software, and even less likely to identify internally developed custom libraries.

- Only libraries with known vulnerabilities can be identified as such.

- Vulnerabilities in code outside the comparison database cannot be reported.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Licensing fees,
    - Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:
    - Annual maintenance fees.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Black Duck Software | Black Duck® Protex™ | http://www.blackducksoftware.com/products/black-duck-suite/protex |
| Palamida | Palamida Enterprise Edition | http://www.palamida.com/products/enterpriseedition |
| Tjaldur Software Governance Solutions | Binary Analysis Tool (BAT) | http://www.binaryanalysis.org/en/home |
| Sonatype | Component Lifecycle Management (CLM) | http://www.sonatype.com/clm/overview |
| OWASP | Dependency-Check | https://www.owasp.org/index.php/OWASP_Dependency_Check |
| Assured Enterprises Inc. | AssuredScanDKV | https://www.assured.enterprises/cyber-products/ |
| Contrast Security | Contrast | https://www.contrastsecurity.com/ |

## 22.  Digital Signature Verification

### a.  Overview

Digital signature verification ensures that software is verified as being from the authorized source (and has not been tampered with since its development).  This typically involves checking cryptographic signatures.

### b.  Details

Digital signatures are used as a mechanism to authenticate data or program files that may be distributed for deployment and use.  For example, most program updates are digitally signed.  Digital signatures help to establish the authenticity, integrity, and non-repudiation of the data or program files provided.

To validate that the content is authentic, has not changed, and is from the authorized originator, a number of things must be ensured.   Is the digital signature valid?  Is the certificate for the digital signature current and not expired?  Is the signer trusted, and is the organization that vouches for the signer (the certificate authority (CA)) reputable?  A common CA for commercial implementations is Verisign; in the case of the DoD, the CA may be the Defense Information Systems Agency (DISA).

A number of automated tools can be used to digitally sign and also to verify the digital signature authenticity.  Digital signatures can be countered by attackers, e.g., an attacker might be able to acquire the private key of an originator or subvert a CA.  Nevertheless, digital signatures enable strong verification.

This is a well-understood mechanism; additional descriptions are widely available, such as in [Microsoft Signature] and [Oracle Signature].

### c.  Applicability

Digital signatures are widely used as a method of verification of authenticity for software update, file transfers, and much more.  It is the standard way to verify that any software updates originate from the expected source.

### d.  Assessment

Digital signing and verification of signature are commodity capabilities that are widely available as open source software or proprietary implementations.  For assurance, digital signing and verification mechanisms are a means for managing software integrity throughout the software development lifecycle (SDLC), especially as acceptance criteria for software delivery.

e. **Resource Requirements**

- Digital signature acquisition

- Valid Certificate authority

- Tools needed to verify the authenticity of the digital signature.

f. **Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Microsoft | SignTool.exe | http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx |
| Symantec | Symantec Code Signing Certificates | http://www.symantec.com/code-signing |
| Oracle | jarsigner - JAR Signing and Verification Tool | http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html |
| E-Lock | E-Lock Reader - Digital Signature Verification Tool | http://www.elock.com/reader.html#.Ua3SM9gr7PM |
| Ascertia | digital signature verification | http://www.ascertia.com/index.aspx |
| OpenSSL Project | OpenSSL | http://www.openssl.org/ |

## 23.  Configuration Checker

a. **Overview**

Configuration checkers assess the configuration of software to ensure that it meets requirements, including security requirements.  A configuration is the set of settings that determine how the software is accessed, protected, and operates.

b. **Details**

Software can have many configuration settings, including ones that impact functionality, security, and safety.  Many configuration settings provide protection or enable services within the software.  Depending on system policy and functional requirements, the configuration may change.

Many tools are available that can check software configurations.  Checking can also be done using manual processes. Either approach typically accesses and analyzes configuration files that are generally part of the software system and vary vastly dependent on the software environment and type of software (e.g., operating system, middleware, or application).

Configuration checkers are related to hardening tools/scripts.  However, hardening tools/scripts can also automatically modify the configurations to improve security and meet policy.

### c. Applicability

Configuration checkers apply to any software.  There are often OTS configuration checkers for OTS software; custom software configurations can also be checked, but someone must determine the rules to be checked.

### d. Assessment

Whether using manual or tools-based configuration checking, this must be performed and periodically reviewed to maintain software assurance.  Poor configuration may lead to unauthorized access by an adversary.  The cost of checking configuration is minimal compared to the potential impact of not doing it.

This approach should balance security and functionality so that adequate protection can be administered while not inhibiting system functionality.

### e. Resource Requirements

Resource requirements are minimal for doing configuration checking.

- Both enterprise and development tools can assist in extracting the relevant information to assess the configuration of the system and its components.

- Training for human resource to complete configuration checking is minimal. Most developers and administrators are hired with this skill intact.

### f. Examples of Suppliers/Products

The following is an example of a suppliers and their product.  This example is provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| GlobalSign | SSL Configuration Checker | https://sslcheck.globalsign.com/en_US |

## 24. Permission Manifest Analyzer

### a. Overview

Permission manifest analyzers are tools that analyze the application's permission manifest and estimate level of risk (possibly using policy requirements to determine what is more or less risky).

### b. Details

This is similar to a configuration checker, in that it examines a small set of data to warn of potential problems. For example, if an application has access to the microphone, audio, network, and location, it typically presents a far greater risk to the user (e.g., as spyware) than an application with none of those permissions. There are many variations, e.g., C-Ray (among other functions it performs) gives a summary of the security posture and permissions on each of the activities exposed by the application.

### c. Applicability

This type of tool/technique requires that there be a permission manifest (e.g., such as Android's). This tool/technique type only examines a small set of data at a gross level, so although it can warn of a few risks, it cannot identify many potential security problems.

Actual tools can include many tool/technique types; this is especially true for this tool/technique type, since the information from a permission manifest may be the basis for other kinds of analysis.

### d. Assessment

Pros:

- Simple and easy to apply.

Cons:

- They do not directly find vulnerabilities, but instead warn of potentially high-risk sets of permissions.

- Cannot find most problems.

### e. Resource Requirements

These have relatively low resource requirements.

**f.  Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| C-Ray Project | C-Ray | https://code.google.com/p/c-ray/wiki/Introduction |

# 25. Development/Sustainment Version Control

**a.  Overview**

Development and sustainment activities are typically supported by version control tools that record and track who made exactly what change and when the change was made.  This information can ease identification of who may have inserted vulnerabilities (unintentional or malicious).  Version control creates a deterrent for inserting vulnerabilities and a starting point for remediation.

**b.  Details**

A variety of tools have been developed to enable multiple developers to work together when revising software and to quickly make changes, record them, merge them, and roll them back if they turn out to be problematic.  Even developers working alone often find such tools helpful.  Such tools are called version control, revision control, and configuration management tools (though in many cases "configuration management" refers to a larger set of processes and goals).

For assurance, version control creates a deterrent for inserting vulnerabilities and a starting point for remediation.  Once a vulnerability is detected, version control enables quick identification of who caused it (the real or the spoofed identity), and when it occurred.  With this information, related changes (e.g., other changes by that individual) can be traced and examined, and if necessary, remediated.

Version control systems are often deployed in ways that presume that SDLC participants are not malicious.  If they could be malicious, the tools must often be additionally configured and/or hardened (e.g., every developer must be assigned a digital signature, and the tool must require these signatures).  It may not be possible to do this with older tools that are not being robustly maintained.

Historically, most version control systems were centralized, that is, they depended on a single central repository that stores the history of changes.  More recently distributed

version control systems (such as "git") have been developed; these enable disconnected operations (e.g., due to lack of network connectivity or because of classification and export control restrictions).

### c. Applicability

These tools are typically used during development and sustainment. They could in some cases be used through operations as well, but this is not as widely practiced; configuration management of operational software or systems is often accomplished by different sets of tools.

Assessors may examine this version control data, when they can obtain it, to assess likelihood and impact of inadequate assurance. For this to be useful for assurance, the version control data must be trustworthy.

### d. Assessment

Pros:

- Such tools are already in use in most organizations, and configuring them to deal with malicious developers is often not a difficult extra step.

- They provide a deterrent against individuals inserting intentional vulnerabilities, and inserting an unusually large number of unintentional vulnerabilities, by making it easy to track down who did it when.

- They simplify triage to identify a single malicious developer, since all the changes that particular developer made can be identified.

Cons:

- Although proprietary COTS developers are likely to use such tools, they often do not wish to reveal much of the data they manage, because this reveals the entire source code suite, as well as much about the organization.

- The tools do not help against a malicious organization, which can forge the version control history.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
  - Licensing fees,
  - Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:
  - Annual maintenance fees,
  - Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Software Freedom Conservancy | git | http://git-scm.com/ |
| Apache Software Foundation | subversion | http://subversion.apache.org/ |
| Perforce Software Inc. | Perforce | http://www.perforce.com/ |
| IBM Rational | ClearCase | http://www-03.ibm.com/software/products/us/en/clearcase/ |
| Microsoft | Team Foundation Server (TFS) | http://tfs.visualstudio.com/ |

Note that many older projects use older version control tools that are not as actively maintained. These older tools include Concurrent Versions System (CVS) and Microsoft Visual SourceSafe (VSS) 2005 (retired from mainstream support on 10 July 2012 with extended support ending on 11 July 2017).

## 26. Obfuscator

### a. Overview

An obfuscator tool takes source, bytecode, or binary and transforms it into something difficult to understand or reverse-engineer. Such tools use approaches to obfuscate code, possibly including the use of reflection and the unnecessary use of native code.

## b. Details

Obfuscators are used to make source, binary or byte code more difficult to understand, decompile, or reverse-engineer. Obfuscators can also be used to make it difficult for applications (including enterprise and mobile) to be changed or manipulated in ways not authorized by the original developer. However, the use of obfuscators also makes it more difficult for downstream users to examine the software to determine their risks.

Mobile application providers may use obfuscators as a way to not only protect their intellectual property but also against the prying hands of potential threat agents. Mobile applications are not only touched by the developer of the applications but also a number of third parties including third-party evaluators, third-party application stores, and users. Obfuscation can be used as a means of protection as the applications traverse the various organizations affecting the application.

## c. Applicability

Obfuscators can be applied to any application, with the caveat that the results are more difficult to examine downstream for risk.

## d. Assessment

Pros:

- Fast ramp-up for use, and cost effective.

Cons:

- Limits transparency by those who need to test the application by using the obfuscated version.

## e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Small investment for licensing.

- Recurring resource requirements:

    − Annual maintenance fees are often minimal to none.

## f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Preemptive Solutions | DASHO | Android | https://www.preemptive.com/solutions/android-obfuscation |
| Arxan | GuardIT | Android | http://www.arxan.com/products/guardit/ |
| ProGuard | ProGuard | Java | http://proguard.sourceforge.net/ |

## 27.  Rebuild and Compare

### a.  Overview

The rebuild and compare technique rebuilds a bytecode or binary from its purported source code, and then determines whether the rebuilt version is equivalent to the bytecode or binary provided.  If it is, then the bytecode or binary corresponds to its purported source code (given certain assumptions).

### b.  Details

Many evaluation approaches examine source code, but such examinations have little value if the source code evaluated does not correspond to the bytecode or binary actually executed.  If the bytecode or binary does not correspond to the source code, then what is evaluated might be unrelated to what is used.

The rebuild and compare technique rebuilds a bytecode or binary from its purported source code, and then determines whether the rebuilt version is equivalent to the bytecode or binary provided.  If they are equivalent, then the integrity of the bytecode or binary is confirmed (presuming that the build environment is not malicious).  Otherwise, someone may have maliciously inserted an attack into the bytecode or binary that is not in the original source code.

A challenge is the amount of information required for this technique.  This technique requires not just the source code of a program (as the term is typically described), but also the build instructions used to create the bytecode or binary, as well as very detailed information about the system being used to build it (e.g., such as the exact version and configuration of all compilers in use).  In many situations there will be differences (e.g., there may be embedded time/datestamps), in which case, all differences must be examined and accounted for, which may require detailed knowledge of the build system.  More information can be found at the reproducible builds website.[11]

---

[11] https://reproducible-builds.org/

### c. Applicability

This technique only applies where there are bytecodes or binaries. If the source code is executed directed (e.g., through an interpreter), this approach is irrelevant.

The technique applies to typical application software, but it cannot be directly applied to development tools to counter malicious subversion. This is because development tools can subvert their own development.[12]

### d. Assessment

Pros:

- The technique provides strong evidence that source code evaluations are justified.

- It is easily automated once the initial correspondence is established.

Cons:

- It requires source code and very detailed information about the build environment.

- In practice, it may require changes to the build environment to establish the detailed build environment information.[13]

- It can be difficult to establish initial correspondence, especially for large legacy systems with many subcomponents.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  - Acquiring detailed build environment information and the tools necessary to reproduce the build environment,

  - Tracking down and repairing any identified differences.

---

[12] This problem – the difficulty of applying the "rebuild and compare" approach to development tools and environments – is sometimes known as the "trusting trust" problem [Thompson1984]. Recent research has identified a related technique, called "diverse double-compiling" (DDC), which can be used to determine whether development tools (such as compilers) correspond to their source code [Wheeler 2009]. Full disclosure: One of the authors of this description named and wrote the defining research on DDC.

[13] These build environment changes may, in fact, have long-term benefits. The build environment may be old and/or depend on old equipment. Updating to a newer build environment may establish detailed build environment information, while also providing a faster build environment with fewer defects.

- Recurring resource requirements:
    - Examining and justifying differences.

### f. Examples of Suppliers/Products

Since this is a process that reuses existing build tools, a list of suppliers/products is not appropriate.

## 28. Assurance Case

### a. Overview

An assurance case is "a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's [security] properties are adequately justified for a given application in a given environment" [IATAC 2007].

### b. Details

A software assurance case is an evidence-based approach that shows, in a well-structured way, how evidence can be combined using various arguments to support key claims. There are a number of standards in various stage of development/completion that can provide a framework for building an assurance case. This framework generally has three major categories that the assurance case is built on: claims, arguments, and evidence. See [NDIA 2008] for a discussion on assurance cases, including how they apply to the DoD System Lifecycle. See also [Rhodes 2009].

Assurance cases have been widely used to develop safety cases. Those with experience in doing safety cases should be able to ramp up quickly to apply assurance cases to security issues.

### c. Applicability

An assurance case can be applied broadly across a system or applied to a specific component. It can also be applied at a high level, as a quick analysis, or applied in depth. The appropriate breadth and depth depends on the criticality of the components being analyzed and the resource limitations (time and money).

Tools are helpful in developing and maintaining large assurance cases.

**d. Assessment**

Pros:

- An assurance case helps organize a variety of evidence into a coherent and traceable justification for important claims.

Cons:

- It requires special training and skills to build an assurance case.

- The tools require licensing, learning, and implementing.

- Systems change; to be useful, an assurance case must be maintained in parallel with the system's evolution.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  − Licensing fees for assurance case tools,

  − Hiring and or training of individuals on security assurance case development.

- Recurring resource requirements:

  − Annual maintenance fees,

  − Continued assurance case development and updates as changes in and to system and critical components occur.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Description | URL |
|---|---|---|---|
| Adelard | ASCE | ASCAD (Claims Arguments Evidence) and GSN (Goal Structuring Notation) | http://www.adelard.com/index.html |
| Praxis | eSafetyCase | GSN (Goal Structuring Notation) | http://www.praxis-his.com/ |

## 29.  Formal Methods/Correct by Construction

### a.  Overview

Formal methods use mathematically rigorous techniques and tools for specification, development, and verification of software and hardware systems [Butler].  We use the term "formal method" in this document synonymously with the term "correct by construction," although there are several different definitions for both terms.

### b.  Details

These techniques use mathematically rigorous techniques and tools, enabling the proofs of claims given certain assumptions.  The term "mathematically rigorous" means that specifications are well-formed statements in a mathematical logic, and that the formal verifications (if any) are rigorous deductions in that logic.

At its most rigorous, the system is completely specific, developed, and verified that it will meet its specifications in all cases.  In practice, this is difficult and expensive to do, especially for non-trivial programs, so various approaches are typically used to reduce the effort necessary.  Three levels are often identified:

- Level 0: A formal specification is created, then a program is informally developed from it. This is sometimes called "Formal methods lite."

- Level 1: Level 0, and then prove some selected properties or perform formal refinement of the specification.

- Level 2: Fully prove claims, which are mechanically checked.

Typically only a specific (especially critical) piece is subjected to this level of rigor.

### c.  Applicability

It is impractical, with today's technology, to apply these approaches after-the-fact to most pre-existing software.  Instead, the software must be developed specifically to support such analysis, enabling the use of such tools.

This means that in practice, formal methods are as much a development process as an evaluation process.  Additionally, formal methods can provide very strong evidence for meeting some technical objectives, but their costs often deter their use.  As a result, although we list them as a tool/technique, we do not list them in the "Software SOAR Matrix" described in Chapter 3.

**d. Assessment**

Pros:

- It provides the strongest evidence available that the software meets the specification.  This is particularly true of level 2.

Cons:

- It is typically costly and time-consuming, particularly at level 2.  Approaches do exist that reduce cost and time, but they may reduce rigor or impose important limitations (e.g., some tools cannot handle dynamically allocated constructs, which limits their applicability).

- It typically requires strong knowledge of discrete mathematics; such expertise is relatively scarce, especially in the United States.

- It requires significant training.  The tools typically require a significant amount of time to master.

- The resulting justifications are only as good as their assumptions.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  - Training, including costs for training, and the time of those receiving the training.  These are often very significant.

- Recurring resource requirements:

  - Usage effort.  This is significant even for level 0, and can increase dramatically at higher levels.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| AdaCore / Altran | SPARK Pro | SPARK | http://www.adacore.com/sparkpro |
| Toccata project | Why3 | Why3 (directly); C, Java | http://why3.lri.fr/ |
| ADT Coq (Action for Technological Development) | Coq | Gallina specification language (Coq-specific) | http://coq.inria.fr/ |

## 30. Network Scanner

### a. Overview

A network scanner (for purposes of this paper) identifies network components (nodes) and network connections (ports) by actively interacting with other network components on the network. Using a network scanner is often a first step in using other tools, such as network vulnerability scanners and intrusion detection systems (IDS), and they are often packaged together.

### b. Details

Network scanners are used for network discovery and port scanning of nodes in the network to provide basic information about them. Network scanners can be used to:

- Identify hosts on the network and establish a network or subnet maps,

- Scan ports for open/closed status and any changes to the port at the time of analysis.

In addition, network scanners can often use this information to:

- Determine operating system characteristics using host detection mechanisms,

- Check version numbers of the applications residing on the hosts.

Network scanners can provide a useful starting set of information about an application's attack surface, before using deeper analysis tools.

A network scanner can only report on currently enabled nodes and ports; a port that is open only sometimes, but not at the time of the scan, will not be reported as open.

### c. Applicability

A network scanner can analyze its network surroundings and provide some detail on the various components on which the software system depends (e.g., their configurations, the operating system it sits on top of, and the surrounding network-accessible

applications that it may have a dependency on).  A scanner can be a quick way to inventory the assets on which a software system depends, helping to produce a quick risk profile of risks due to changes in configuration or version changes.

A scanner can help provide a summary view of a system or application's network attack surface.  However, it will typically provide little insight into intentionally malicious behavior, since such software can simply wait to open a port at some future time.

These tools can only provide basic information about the node and port (e.g., the type of operating system in use and the port).  For example, a web server often has port 80 open; a network scanner could report this, but not by itself determine whether vulnerable applications are accessible on that port.  That said, a network scanner is often paired with other functionality that can do deeper analysis.  Some tools bundle in this functionality as a first step in applying other analysis approaches.

### d.  Assessment

Pros:

- These tools are helpful in quickly finding network nodes and ports.

- They enable review of possible access points for breach or policy violations.

Cons:

- They provide limited knowledge; by themselves they only provide information focused at network and node inventory (they are often paired with other functionality for deeper analysis).

### e.  Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Training, including costs for training, and the time of those receiving the training.  Practically using these tools requires some knowledge to understand the tool results.

- Recurring resource requirements:

    − Reviewing periodic results.

### f.  Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Nmap.org | nmap | http://nmap.org |
| Gibson Research Corporation (GRC) | Shields up (online port scanning service) | https://www.grc.com/x/ne.dll?bh0bkyd2 |

## 31.   Network Sniffer

### a.   Overview

A network sniffer (also called a packet analyzer) observes and records network traffic.  This information can then be analyzed to identify unexpected network traffic, perform trend analysis, and so on.

### b.  Details

A network sniffer monitors packets flowing over network/subnets, examining network traffic by making a copy of the data for analysis but without redirecting or altering it.

It can be used for analyzing network problems such as bandwidth utilization, unusual and unusual amounts of traffic, network intrusion attempts, misuses in the network by both internal and external users, filtering of suspect content from packets, and unwanted "call home" functionality. It can also collect login data/user cookies for further analysis.  Network sniffers sometimes detect characteristics that may indicate the potential for man-in-the-middle attacks, e.g., the lack of Secure Socket Layer/Transport Layer Security (SSL/TLS) authentication, poor keys, or poor encryption algorithms.

### c.  Applicability

Sniffers apply to software that produces or consumes network traffic.

### d.  Assessment

Although sniffers are not the first tools one thinks of when addressing assurance, sniffers by themselves provide some data regarding network traffic, and perhaps some information regarding the potential impact on the end system consuming the traffic. These tools provide a way to analyze potential intrusion attempts, including suspect content originating from or destined to a software system/application.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

- Recurring resource requirements:

    − Annual maintenance fees,

    − Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine anomalies in network traffic.

### f. Examples of Suppliers/Products

The following are examples of products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Colasoft | Capsa Network Analyzer | http://www.colasoft.com/capsa/ |
| Massimiliano Montoro | Cain and Abel (password recovery tool for Microsoft operating systems, includes sniffing) | http://www.oxid.it/cain.html |
| Dug Song | dSniff | http://www.monkey.org/~dugsong/dsniff/ |
| Telerik | Fiddler | http://www.telerik.com/fiddler |
| Microsoft | Microsoft Message Analyzer | http://www.microsoft.com/en-us/download/details.aspx?id=40308 |
| Sky software | SkyGrabber; LanGrabber | http://www.skygrabber.com/en/index.php |
| Oracle | snoop (in Solaris) | http://docs.oracle.com/cd/E23824_01/html/821-1453/gexkw.html |
| Tcpdump (project) | tcpdump | http://www.tcpdump.org/ |
| Wireshark (project) | Wireshark (formerly known as Ethereal). | https://www.wireshark.org/ |

As of 2014-03-10, the following web pages list a number of network sniffers: http://en.wikipedia.org/wiki/Packet_analyzer and http://sectools.org/tag/sniffers/.

## 32. Network Vulnerability Scanner

### a. Overview

For the purpose of this paper, a network vulnerability scanner examines a system through its network interface (e.g., its network ports) to identify known vulnerabilities.

### b. Details

A network vulnerability scanner sends messages to the various network ports of a system and examines the results to determine whether the system being examined has any known vulnerabilities. In particular, a scanner attempts to identify services whose implementation has known vulnerabilities (e.g., an obsolete web server with a known vulnerability) and any indicators of an insecure configuration.

A network vulnerability scanner may be used in tandem with other tools. For example, a network vulnerability scanner may be used after a network sniffer and scanner identifies the system for further analysis. If a network vulnerability scanner does not identify a known vulnerability, an application-type-specific vulnerability scanner may search for vulnerabilities that are not already known. Tool implementations may combine techniques.

More information about network vulnerability scanners is available in a variety of places, including [Guirguis 2003] and [HKSAR 2008].

### c. Applicability

These tools require network access to the software being evaluated.

### d. Assessment

Pros:

- These tools are helpful in quickly finding well-known vulnerabilities.

- They only require network access; not even executables are required.

Cons:

- They can only find already-known vulnerabilities.

- Many tools produce false positives if they merely report on indicators instead of actually trying to perform an exploit. For example, the tool may report that a service is vulnerable if the program is misleadingly reporting the wrong program

or version name, or if the program has been configured in a different way than expected to counter the vulnerability.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed,

  − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  − Annual maintenance fees.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Tenable Network Security | Nessus | http://www.tenable.com/products/nessus |
| OpenVAS project | OpenVAS | http://www.openvas.org/ |
| Rapid7 | Nexpose | http://www.rapid7.com/products/nexpose/ |
| Rapid7 | Metasploit | http://www.metasploit.com |

## 33. Host-based Vulnerability Scanner

### a. Overview

For the purpose of this paper, a host-based vulnerability scanner examines a host system configuration for flaws and ensures that the host configuration meets certain predefined criteria. It may also verify that the audit mechanisms work. This type of tool can be used both before deployment and during operations.

**b. Details**

A host-based vulnerability scanner is similar in some ways to a network vulnerability scanner, but at least part of its functionality is on a host system. Thus, it has much more access to information on the host system.

Host-based vulnerability scanners typically focus on identifying (and/or countering) known problems. They "are able to recognize system-level vulnerabilities including incorrect file permissions, registry permissions, and software configuration errors. Furthermore, they ensure that target systems are compliant with the predefined company security policies. Unlike network-based scanners, an administrator account or an agent is [typically] required to be on the target system to allow for the system-level access required." [Guirguis 2003]

This type of tool can be used both before deployment and during operations.

Some tools are focused on specific types of programs being analyzed, e.g., database analysis; these are discussed separately. In practice they may be coupled with other tools, such as network-based vulnerability scanners.

More information is available in a variety of places, including [Guirguis 2003] and [HKSAR 2008].

**c. Applicability**

At a minimum, these tools must have access to the host files (especially its configuration files), and typically must be allowed to execute program(s) on the host.

**d. Assessment**

Pros:

- These tools are helpful in quickly finding well-known vulnerabilities.

Cons:

- They must have direct access to the host system being analyzed, or at least the host system data.

- Many tools produce false positives if they merely report on indicators instead of actually trying to perform an exploit. For example, the tool may report that a service is vulnerable if the program is misleadingly reporting the wrong program or version name, or if the program has been configured in a different way than expected to counter the vulnerability.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

    − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

    − Annual maintenance fees.

### f. Examples of Suppliers/Products

Note that host-based scanning is often integrated in with other functionality, e.g., network-based scanning.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|----------|--------------|-----|
| Microsoft | Microsoft Baseline Security Analyzer | http://technet.microsoft.com/en-US/security/cc184924.aspx |

## 34. Host Application Interface Scanner

### a. Overview

For the purpose of this paper, a host application interface scanner identifies the various host-based interfaces of applications.

### b. Details

A host application interface scanner at a minimum enumerates the various host-based interfaces for applications. For example, on an Android platform, such programs should identify the activities, broadcast receivers, content providers, and services. Such interface scanners may also report other information about the applications (such as privileges granted to them). Finally, such tools may also be able to create messages to those interfaces (e.g., to perform penetration testing).

### c. Applicability

These tools may be especially useful in penetration testing. Expertise is required to understand the outputs of these tools.

### d. Assessment

Pros:

- Provide insight into applications.

Cons:

- Require significant expertise.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

  − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  − Annual maintenance fees.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| MWR | Drozer | https://www.mwrinfosecurity.com/products/drozer/ |

## 35. Web Application Vulnerability Scanner

### a. Overview

For purposes of this paper, a web application vulnerability scanner automatically scans web applications for potential vulnerabilities. These tools simulate a web browser

user, dynamically trawling through URLs and trying to attack the web application.  For example, they may perform checks for field manipulation and cookie poisoning [SAMATE].

### b.  Details

For our purposes, a web application vulnerability scanner focuses on dynamic analysis, simulating a web browser to look for vulnerabilities in a system.  For static analysis (analyzing the source code or executable), see the other tool/technology types. Note that some analysis tools combine static and dynamic analysis techniques to find vulnerabilities in web applications.

In some cases these scanners create attack input to see whether the web application can counter the attack (often by using specialized fuzz testing techniques).  For example, to determine whether a web application is vulnerable to SQL injection, it may create and send data specifically designed to try to trigger SQL injection attacks (such as inserting single quotes with SQL commands that follow).  As a result, such tools can find unknown vulnerabilities in web applications.  These tools will focus on known types of vulnerabilities (a.k.a. "weaknesses"), not on radically new types of vulnerabilities, but they can still be useful since most vulnerabilities are of common types.

NIST Special Publication 500-269 proposes "a minimum (mandatory) level of functionality in order for the purchaser and vendor to qualify" for a "web application security scanner specification," their terminology of the time [Black 2008].  It includes a list of web application vulnerabilities that such a tool should specifically look for, such as SQL injection.  Note that not all web application vulnerability scanners will necessarily meet this level of functionality, and certainly not all scanners will meet that functionality equally.

More information is available in a variety of places, including [Guirguis 2003] and [HKSAR 2008], as well as the SAMATE page on web application vulnerability scanners http://samate.nist.gov/index.php/Web_Application_Vulnerability_Scanners.html.\

Shay Chen has an extensive review of these kinds of tools [Chen 2014]. In his report he recommends that "when trying to figure which tool you should use, try the following simple methodology":

1.  Input Vector and Scan Barrier Support. "Figure out if the input delivery method used by the application or applications you are using is supported by the scanners you are evaluating. Do the same for the various security mechanisms, technologies and scan barriers that are used in the application (Text X). The scanner won't work at all, or will provide little value if it won't support those."

2.  Crawling & Input Vector Extraction.  "If you use scanners mainly in a point-and-shoot scenario, and prefer as much automation as possible, a [good automated

crawler] will be the second most important feature you should follow." Chen suggests using a high Web Input Vector Extractor Teaser (WIVET) score as a way to measure this.

3.  Vulnerability Detection Features and Accuracy. Prefer tools with good vulnerability detection features and accuracy.

4.  Price.

### c. Applicability

These tools only apply to web applications, and they require dynamic (network) access to the software being evaluated.

Some suppliers provide this capability *only* as an external service.

### d. Assessment

Pros:

- These tools are helpful in quickly finding certain kinds vulnerabilities, including vulnerabilities not known to the developers.

- They are relatively easy to get started if a test system is already available.

Cons:

- They must be able to execute the web application.

- Depending on how they are implemented, using these tools can corrupt the underlying system's data or interfere with its operation if applied against a production system. Some tools are designed to minimize this risk, but that is still a potential concern. This can be countered by applying the tools to a test machine instead, but such a system must be made available, and test systems often differ from production systems in important ways.

- It takes time to track back from a vulnerability discovery to determine what the problem is and how to fix the problem (this is a difficulty with almost any dynamic tool).

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    - Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

- Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:
  - Annual maintenance fees.

### f.  Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL | License |
|---|---|---|---|
| IBM | IBM Security AppScan Standard | http://www-03.ibm.com/software/products/us/en/appscan-standard | Proprietary |
| Wapiti project | Wapiti | http://wapiti.sourceforge.net/ | OSS |
| W3af project | W3af | http://w3af.org/ | OSS |
| PortSwigger Ltd | Burp Suite | http://portswigger.net/burp/ | Proprietary |
| White Hat Security | SecurityChec k | https://www.whitehatsec.com/ | Proprietary |
| OWASP ZAP project | Zed Attack Proxy (ZAP) | https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project | OSS |
| HP | HP WebInspect | http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/index.html?.Uqi69_RDs7c | Proprietary |
| Netsparker LTD | Netsparker | https://www.netsparker.com/ | Proprietary |
| Arachni | Tasos Laskos | http://www.arachni-scanner.com/license/ | OSS |
| Acunetix | Acunetix WVS | http://www.acunetix.com/vulnerability-scanner/ | Proprietary |
| NT OBJECTives | APPSPider (fka NTO Spider) | http://www.rapid7.com/products/appspider/capabilities.jsp | Proprietary |
| Qualys | QualysGuard Web Application Scanning (WAS) | https://www.qualys.com/enterprises/qualysguard/web-application-scanning/ | Proprietary |

The web application security consortium (WASC) has a list of web application security scanners at http://projects.webappsec.org/w/page/13246988/Web%20Application%20Security%20Scanner%20List, and OWASP has a list of tools at https://www.owasp.org/index.php/Phoenix/Tools.  Also, [Chen 2012] [Chen 2014] presents test results for a large number of web application vulnerability scanners.

## 36.  Web Services Scanner

### a.  Overview

For the purpose of this paper, a web services scanner automatically scans a web service (as opposed to a web application), e.g., for potential vulnerabilities. [SAMATE]

### b.  Details

A web services scanner focuses on dynamic analysis, simulating a client to look for vulnerabilities in a system.  For static analysis (analyzing the source code or executable), see the other tool/technology types.  Note that some analysis tools combine static and dynamic analysis techniques to find vulnerabilities in web applications.

Fundamentally, a web services scanner is very similar to a web application vulnerability scanner, but it is focused on web services instead of web applications.

More information is available in a variety of places, including [Guirguis 2003] and [HKSAR 2008], as well as the SAMATE page on web services network scanners http://samate.nist.gov/index.php/Web_Services_Network_Scanners.html.

### c.  Applicability

These tools only apply to web services and require dynamic (network) access to the software being evaluated.

### d.  Assessment

Pros:

- These tools are helpful in quickly finding certain kinds vulnerabilities, including vulnerabilities not known to the developers.

- They are relatively easy to get started if a test system is already available.

Cons:

- They must be able to execute the web service.

- Depending on how they are implemented, using them can corrupt the underlying system's data or interfere with its operation if applied against a production system. Some tools are designed to minimize this risk, but that is still a potential concern. This can be countered by applying the tool to a test machine instead, but such a system must be made available, and test systems often differ from production systems in important ways.

- Time is needed to track back from a vulnerability discovery to determine what the problem is and how to fix the problem (this is a difficulty with almost any dynamic tool).

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

  − Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  − Annual maintenance fees.

### f. Examples of Suppliers/Products

The following is an example of a supplier and their product. This example is provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|----------|-------------|-----|
| IBM | IBM Security AppScan Standard | http://www-03.ibm.com/software/products/us/en/appscan-standard |

## 37. Database Scanner

### a. Overview

For the purpose of this document, a database scanner is "a specialized tool used specifically to identify vulnerabilities in database applications" [SAMATE]. For

example, they may detect unauthorized altered data (including modification of tables) and excessive privileges.

### b. Details

A database scanner does a detailed security analysis of database systems. This includes authentication, authorization, and integrity of the database systems. It can also identify potential security exposure in a database system, such as weak passwords, security misconfigurations, and (in some cases) Trojan horses.

Many applications (including web applications) build on top of a database. Thus, a tool that focuses on its database usage may identify problems, even if the database scanner has no specific knowledge about that application. This means that database scanners can detect previously unknown application vulnerabilities, but only if they relate to how they use their database.

More information is available in a variety of places, including [Guirguis 2003] and [HKSAR 2008], as well as the SAMATE page on database scanning tools (scanners) at http://samate.nist.gov/index.php/Database_Scanning_Tools.html.

We categorize this as "dynamic" because these tools often run the database program to gather the information they analyze. However, this is not always so, and the specific data that they analyze is often static in nature. Some database scanners are host-based, but this is not necessarily so.

### c. Applicability

These tools only apply to databases, including applications that use databases.

### d. Assessment

Pros:

- These tools are helpful in quickly finding certain kinds vulnerabilities, including vulnerabilities not known to the developers.

- They are relatively easy to get started.

Cons:

- They only report on database issues, which is typically only a portion of an application.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

- Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.

- Training, including costs for training, and the time of those receiving the training.

- Recurring resource requirements:

  - Annual maintenance fees.

### f.  Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Imperva | Scuba | http://www.imperva.com/products/dsc_scuba-database-vulnerability-scanner.html |
| Security Wizardry | DBAPPSecurity Database vulnerability scanner(DB-Scan | http://www.securitywizardry.com/index.php/products/scanning-products/database-scanners/database-vulnerability-scanner.html |

## 38. Fuzz Tester

### a.  Overview

A fuzz tester presents software systems with test cases that are invalid, unexpected, or random, as a testing mechanism to determine whether problems occur (e.g., crashes, failed operations, or memory leaks).  Fuzz testing technology can be used as a standalone function and is integrated into various other tools (e.g., scanners for creating denial of service).

### b.  Details

**Fuzz testing** or **fuzzing** is an automated black box testing technique for software that involves providing invalid, unexpected, or random data to the inputs of a computer program.  The software under test (SUT, which is also the TOE) is monitored for crashes, failure of operations, and other problems (such as memory leaks, use of unallocated memory [Serebryany 2012], or assertion failures).  Tools that implement fuzz testing are called fuzz testers or fuzzers.

There are several different dimensions to consider when selecting a fuzzer. These include how it generates the test cases, the target (aka attack vector) that it can test, and what feedback it can use.

There are many different ways to generate test cases. Approaches include:

- Random. Here the software selects random values with no feedback from or knowledge about the SUT. The fuzz test inputs can be truly random, or they may be tailored to increase the likelihood of detecting certain kinds of defects. For example, they may be tailored to include text fragments more likely to cause a SQL injection, or may include context-specific inputs more likely to cause a crash.

- Mutation. Mutation-based fuzzers mutate existing data samples to create test data, e.g., by flipping bits or moving blocks around.

- Specification. Specification-based fuzzers (aka generational, model-based, protocol-based, or "smart" fuzzers) are provided a specification (a model) of the expected input and use this to generate input data by adding anomalies. Creating these specifications takes more time but may provide more in-depth results. These can be challenging to apply if the specification is unknown.

One research paper stated that, "The advantage to mutation-based fuzzing is that little or no knowledge of the protocol or application under study is required... [while] generation-based fuzzing requires a significant amount of up-front work to study the specification and manually generate test cases. Regardless, intuition says that the extra knowledge gained by understanding the format should result in higher quality test cases." They then performed an experiment that confirmed this; they measured the amount of executed code required to parse PNG image files, and in their case found that "generation-based fuzzing can execute 76% more code when compared to mutation-based methods." [Miller2007]

Any of these generation approaches can also support evolution; that is, accepting feedback from the SUT to make better decisions about future test cases. In some cases, routines such as checksum checkers may need to be disabled or specially handled when generating test cases (particularly if the fuzzer is not a specification-based fuzzer).

Fuzzers also vary in the kinds of targets they support. They may be designed to test only applications that use specific frameworks, environments, or protocols. In some cases specialized fuzzers are created for a specific program. Therefore, it is important to check whether the fuzz tester being considered for use would be appropriate for the software it would be testing. Different fuzzers support fuzzing of file contents, file systems, environment variables, APIs, and/or network protocols at various levels. These may be divided in two classes:

a. Interactive Fuzz testers. Interactive fuzzers support applications that require interaction via some kind of protocol, such as a network protocol (e.g., FTP) or web application (e.g., HTTP).

b. Non-interactive Fuzz testers. Non-interactive fuzzers support applications that do not require interactive protocols, such as file formats or environmental variables.

Fuzzers also differ in the information they can use from execution. Nearly all fuzzers at least note when a program crashes, interpreting that as a potential problem. Some fuzzers (particularly if they are specification-based) can examine responses to determine whether those results are sensible (a fuzzer will be able to identify when a response is outside a permitted range). Heartbleed, for example, was found by Codenomicon using this technique [Wheeler2015]. In addition, other tools may be used to help detect problems, e.g., an address sanitizer may be used to detect when a program accesses memory it should not be accessing [Serebryan2012] [Böck]. Since earlier SOAR reports, there has been an improvement in the information fuzzers often use to detect problems, leading to improvements in fuzzing results (e.g., read/write outside of buffers, including buffer overflows, is more likely to be detected when these improved sources of information are used).

Some fuzzers leverage code coverage to indicate which areas need further examination; see "coverage-guided fuzz tester" (section C.55).

There are many other ways to categorize fuzzers; this is an active area of research and development.

Fuzzers can work well in identifying problems that might cause a program to crash, such as buffer overflow, denial of service attacks, format bugs, input validation errors, and SQL injection, which are often used by malicious attackers to cause the largest impact using the fewest possible resources. Fuzz testing is often less effective for dealing with security threats that do not cause program crashes, such as spyware, some viruses, worms, Trojans, and key loggers.

Fuzz testing can often reveal defects that are overlooked when software is written and debugged. However, fuzz testing usually finds only the most serious faults. It cannot be used to provide a complete picture of the overall security, quality, or effectiveness of a program in a particular situation or application. Also, fuzz testing is often subject to diminishing returns; once initial problems are fixed, fuzz testing can be progressively less effective at finding more. Fuzzers are most effective when used in conjunction with extensive black box testing, beta testing, and other methods.

Fuzz testing is a very general technique. Therefore, fuzz testing approaches may sometimes be included as part of other tool types. For example, web application

vulnerability scanners typically incorporate fuzz testing; for more information, see section C.34 above.

Books that provide more detail about fuzzing include [Sutton2007] and [Takanen2008].

### c. Applicability

Fuzz testing is a widely applicable testing technique for software, because it is simple and may offer a high benefit-to-cost ratio.

### d. Assessment

Pros:

- Fast ramp-up for use,

- Fast method of identifying some critical issues in software systems,

- Extremely cost effective approach.

Cons:

- Not a comprehensive testing approach,

- Only certain types of security flaws would typically be discovered by fuzz testing,

- Typically has diminishing returns.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  - Small investment for licensing (fuzz testers are often inexpensive compared to other analysis tools).

- Recurring resource requirements:

  - Annual maintenance fees are often minimal to none.

### f. Examples of Suppliers/Products

Many programs implement fuzz testing. In addition, some interviewees stated that they found it just as effective to write specialized fuzzing programs, since they are fairly easy to write and specialized fuzzers can target properties of specific applications and platforms.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Beyond Security | beSTORM | Binary, in-memory | http://www.beyondsecurity.com/bestorm.html |
| Synopsys Codenomicon | Defensics | Windows, Linux | http://www.codenomicon.com/products/defensics/ |
| Microsoft Research | Sage | .NET platform | http://research.microsoft.com/en-us/um/people/pg/public_psfiles/cacm2012.pdf |
| Peach Fuzzer | Peach Fuzzer Platform | .NET, Python | http://www.peachfuzzer.com/ |
| Caca labs | zzuf | POSIX (including Linux) | http://caca.zoy.org/wiki/zzuf |
| CERT | CERT fuzzers: Basic Fuzzing Framework (BFF) and the CERT Failure Observation Engine (FOE). | BFF is for Linux and Mac OS; FOE is for Windows | https://github.com/CERTCC-Vulnerability-Analysis/certfuzz |

## 39. Framework-based Fuzzer

### a. Overview

For our purposes, a framework-based fuzzer creates inputs and observes results, as with traditional fuzzing, but it instruments the underlying platform framework to help identify and select what inputs would be most relevant to test.[14]

### b. Details

One challenge of using traditional fuzzing is that it does not have information on the internals of a program. As a result, it often fails to test significant portions of a program, resulting in a "shallow" test.

However, many applications are built using a common framework, which typically includes various "registration" facilities to identify what is important to the application. For example, in mobile applications, "A distinctive aspect of mobile apps is that all such

---

[14] Note that the term "framework-based fuzzers" has multiple different meanings in the literature.

apps, regardless of how diverse their functionality, are written against a common framework that implements a significant portion of the app's functionality" [Machiry]. The same can be said for many web applications (though for web applications there is a large set of frameworks, not a single one).

An application can use platform framework information to focus its test generation, and also use the framework to extract information on the success of the test. This additional information can help improve the depth of testing, and in particular, may help improve security-relevant testing. For example, a framework-based fuzzer may build in knowledge of what sequences or states must not occur and what input patterns are more likely to cause security breaches.

### c. Applicability

An application framework is necessary for this approach. The more widespread the framework, the more useful a particular testing tool to use the framework can be. Mobile applications are typically built as extensions of a preset framework provided by the mobile operating system, making it easier to build one tool that can apply to many different applications.

### d. Resource Requirements

Pros:

- Typically does not require source code of applications.

Cons:

- Based on recent research, so they are less mature, and their limitations are less understood

- Only certain types of security flaws would typically be discovered by fuzz testing.

### e. Resource Requirements

Resource requirements are uncertain at this time, and probably vary depending on the framework and context applied.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Frameworks Supported | URL |
|---|---|---|---|
| Georgia Institute of Technology | Dynodroid | Android | http://pag.gatech.edu/dynodroid/ |

## 40. Negative Testing

### a. Overview

For the purpose of this document, negative testing is the technique of including in the regression test suite many tests that should *fail* if the security mechanisms work properly. This technique is typically implemented as a test case generation criteria when using existing test tools.

### b. Details

Many developers are understandably focused on ensuring that a system works correctly when given correct inputs. This can lead to an unfortunate blindness; developers may fail to ensure that a system works when given incorrect or malicious inputs, including inputs that should fail because of security and input validation mechanisms.

If a software system is properly developed there will be an automated regression test suite that can re-test the software to check whether it is working correctly. Such automated regression tests should be run often (e.g., every build or every week). Negative testing simply requires an automated regression test suite that includes tests of the security mechanisms. This includes testing the input validation mechanisms and preventing accesses and requests that should be prevented. Vulnerabilities previously fixed should trigger the creation of new tests in the automated regression test suite, to prevent later changes from re-introducing the vulnerability.

### c. Applicability

Any software development organization can perform negative testing. In particular, they should already be maintaining and using a regression test suite, so this is primarily a matter of training and funding testers to include negative tests in the test suite. The recurring costs in many cases would be absorbed into the costs of developing the regression test suite.

This approach could in theory be performed by those outside the software development organization, e.g., by potential users. A regression test suite framework for the component would need to be established and maintained; doing this outside the development organization would be more difficult.

**d. Assessment**

Pros:

- Negative testing can be easily integrated into the existing regression test suite.

Cons:

- It requires training; many developers and testers do not have experience creating these kinds of tests.

- It requires effort to create the initial tests.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

    – Training to create these kinds of tests,

    – Cost of creating initial tests.

**f. Examples of Suppliers/Products**

Since this is a process that reuses existing tools, a list of suppliers/products is not appropriate.

# 41. Digital Forensics

**a. Overview**

For the purpose of this paper, digital forensics tools are tools that support "the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations," per the Digital Forensics Research Workshop [Carrier 2003] [Palmer 2001].

**b. Details**

Applications can be executed and the resulting stored data can then be analyzed to learn important facts about the application. For example, preset sensitive data (including credentials) can be entered into an application, and then forensics tools can capture the stored data to see if it was properly encrypted or otherwise protected.

The NIST Computer Forensics Tools Testing (CFTT) program provides a measure of assurance that the tools used in the investigations of computer-related crimes produce valid results [NIST CFTT].

### c.  Applicability

Forensics tools can reveal failures to perform credential encryption and some exposures of sensitive information.

### d.  Assessment

Pros:

- Can quickly obtain stored data/evidence for examination.

Cons:

- Its Utility in evaluating applications is limited.

### e.  Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Training to create these kinds of tests,
    - Cost of creating initial tests.

### f.  Examples of Suppliers/Products

Since this is a process that reuses existing tools, a list of suppliers/products is not appropriate.

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Frameworks supported | URL |
|---|---|---|---|
| viaForensics | viaLab (portion) | Android | https://viaforensics.com/products/vialab/ |

# 42. Intrusion Detection Systems/Intrusion Prevention Systems

## a. Overview

An Intrusion Detection System (IDS) monitors network or system activities for malicious activities or policy violations and reports them. An Intrusion Prevention System (IPS) also monitors, but instead of just reporting activities or violations, it actively prevents or remediates them. There are two major types of IDS/IPS:

- Network-based IDS/IPS. A network-based IDS or IPS monitors network traffic to perform its monitoring, prevention, and/or remediation for malicious activities or policy violations.

- Host-based IDS/IPS/Integrity checker. A host-based IDS, IPS, or integrity checker monitors data other than network traffic (such as files, registry values, and program input/output) for malicious activities or policy violations.

## b. Details

Intrusion detection and intrusion prevention systems monitor network traffic and or system events for malicious activities. IDSs tend to be passive; they may be placed in parallel or in line, monitoring events and traffic and alerting about suspicious events, changes or anomalies. IPSs are often placed in line and actively prevent or block intrusions, including shutting off ports or blocking traffic.

IDS/IPS systems (both network and host-based) can be divided into signature-based and statistically-based approaches:

- Signature-based detection uses attack patterns or expected usage patterns that are preconfigured and predetermined. As traffic traverses the network, an IPS/IDS analyses the traffic for a signature match (or non-match) and delivers an alert to the appropriate system or person and in many cases, takes preventative action as well.

- Statistical anomaly-based detection initially takes a baseline or average of network traffic conditions. It then samples the network traffic to compare current traffic against this baseline to verify that activities are within expected parameters.

For more information see [Holland 2004].

## c. Applicability

IDS/IPS systems are useful in the operational environment because they can detect or prevent potential intrusions/breaches. This enables countering intrusions/breaches directly, as well as possibly suggesting future changes to the system to make future

attacks even more difficult to perform. Similarly, IDS/IPSs are useful in the development and sustainment environment, because they can sometimes detect or prevent potential intrusions/breaches aimed at subverting the software in development or sustainment.

For software systems assurance, IDS/IPSs can be used to evaluate software by running the software under its monitoring. This may occur during more extensive processes such as monitored execution and penetration testing. The IDS/IPS data may then be used to help determine level of risk. For example, if the IDS/IPS reports a number of suspicious activities, those can be investigated or simply used as evidence to prefer a different product.

### d. Assessment

Pros:

- The technology is already widely adopted and used.

- It enables data collection.

Cons:

- It can only detect problems if it can observe the problematic state or behavior *and* also have signatures or statistics that suggest that they are problems.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    - Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

- Recurring resource requirements:

    - Annual maintenance fees,

    - Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
| --- | --- | --- | --- |
| Cisco | Intrusion Prevention System (IPS) | Network, host based intrusion prevention solutions – integrated into FW or separate | http://www.cisco.com/en/US/products/ps5729/Products_Sub_Category_Home.html |
| Cisco (formerly Sourcefire) | SNORT | free and open source network intrusion prevention system | http://en.wikipedia.org/wiki/Snort_%28software%29 |
| Open Information Security Foundation (OISF) | Suricata | Free open source Network IDS, IPS and Network Security Monitoring engine | http://suricata-ids.org/ |

## 43.  Automated Detonation Chamber (limited time)

### a.  Overview

An automated detonation chamber (limited time) automatically isolates a program and/or data (including running multiple copies in virtual machines), executes/processes it, detects potentially malicious or unintentionally vulnerable activities, and then reports its findings (typically prior to the software's deployment).

### b.  Details

A detonation chamber is a system that runs a program or processes data in isolation so that its behavior during execution can be monitored.  This monitored execution can vary from a quick run-through using a single tool to a fairly extensive analysis using multiple tools/techniques to detect intentional and unintentional malicious/unplanned behavior.  Using a detonation chamber for monitored execution is general approach that can combine many different approaches, as discussed in the main body of this paper.

This type of tool typically focuses on executing the software under test for a limited time to monitor execution to detect malicious behavior.  These tools often execute programs in virtual environments to make it easier to observe results such as registry changes, file changes, creation of intents, etc.  This technique can be performed using many parallel executions for any one application, observing for potentially unwanted behavior across many possible input spaces.  Often the limited time is fixed, but in some cases the time may be extended (e.g., if more suspicious or risky behavior is observed).  This technique can also be performed for multiple applications simultaneously.

Many formats typically considered data (such as portable document format (PDF) and the office file formats) can include executable code or data that could exploit the

processing program. Thus, this approach can be applied to formats typically considered as data.

Note that this is different from the approaches used by a virus scanner, intrusion detection system (IDS) or an intrusion prevention system (IPS). In an automated detonation chamber the potentially malicious software is not run in the final environment. Since it is not running in the final environment, if the software is malicious, it cannot cause the kind of damage as it would if it were installed in its final location.

Previous versions of this document referred to this approach as "automated monitored execution (limited time)." However, the term "automated detonation chamber" seems to be more commonly used and is clearer. In particular, it is possible to monitor applications as they run in the real environment, which is different from the approach described here.

### c. Applicability

This can potentially apply to any application. This approach can be used today as an inline analysis technique in network infrastructure, e.g., it can be used to monitor email attachments for malicious behavior prior to allowing the attachments to reach their destination.

This approach is often used in operational settings. In these cases, if malicious indicators are detected, that information is often sent out to other operational systems (including the supplier's or other customers). This automated sharing can help prevent others from running the malicious code or data.

Note that one potential use for an automated detonation chamber can be to determine excessive power consumption (e.g., for mobile devices). Excessive power consumption can directly indicate attacks focusing on draining mobile device power (as a denial of service attack). Excessive power consumption, or excessive CPU utilization, may also be an indicator of some kinds of intentional malware that is stealing processing cycles (e.g., to mine virtual currencies like Bitcoin).

### d. Assessment

Pros:

- Fast ramp-up for use,

- Fast method of detecting some malicious behavior before the application is used.

Cons:

- Not a comprehensive testing approach.  In particular, applications that delay malicious activities may not be detected

- Dependent on tool vendor algorithms for learning and administering heuristics.

### e.  Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Investment for licensing.

- Recurring resource requirements:

  − Annual maintenance fees are often minimal to none.

  − Training for tool usage.

### f.  Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| FireEye | Virtual execution engine | Selected document formats on Windows | http://www.fireeye.com/products-and-solutions/virtual-execution-engine.html |
| Palo Alto | Wildfire | Selected document formats and websites on Windows and Android | https://www.paloaltonetworks.com/products/secure-the-network/subscriptions/wildfire and https://www.paloaltonetworks.com/resources/datasheets/wildfire |
| Cuckoo Foundation | Cuckoo Sandbox | Files and websites on Windows, OS X, Linux, and Android | https://www.cuckoosandbox.org/ |
| Appthority | App risk management service | Android, iOS | https://www.appthority.com/products/service-works |
| Veracode | Veracode App Intelligence (vAI) (partial) | Android, iOS | https://www.veracode.com/products/application-security-analytics.html |

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| DroidBox project | DroidBox | Android | http://code.google.com/p/droidbox/ |
| Kryptowire | Kryptowire | Android, iOS | http://www.kryptowire.com/ |

## 44. Forced Path Execution

### a. Overview

Forced path execution runs a program and forces execution of all (control flow) paths, even if the test inputs would not normally cause it to do so, and monitors what happens to detect possible undesired behavior.

### b. Details

Most dynamic execution approaches only test a portion of the program, namely, the parts exercised based on the inputs given. In this approach, the other paths are forced to be executed anyway, to see what happens to the data on those branches. For example, in a mobile environment, it could examine to see whether sensitive data (such as contact information, microphone sensor data, or camera data) is stored in or transmitted to unexpected destinations, or if some data is not encrypted in some cases.

This approach does raise a potential risk of false positives, but it also increases program coverage within reasonable scale. This approach can be easily scaled by applying multi-core systems to execute the various control paths simultaneously, reducing the time for analysis to complete.

### c. Applicability

This may be especially helpful for identifying some kinds of exfiltration or other exposures of sensitive data.

### d. Assessment

Pros:

- Fast method of detecting some malicious behavior before the application is used,

- More code coverage than typical for a dynamic analysis tool/technique.

Cons:

- Potential for significant false positives.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    – Investment for licensing.

- Recurring resource requirements:

    – Annual maintenance fees are often minimal to none.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Kryptowire | Kryptowire (Android edition)[15] | Android bytecode | http://www.kryptowire.com/ |

## 45. Firewall

### a. Overview

A firewall limits network access based on a set of rules. A firewall can be network-based (e.g., used as a gateway into a network) or host-based (limiting access between one host and a network). They typically check traffic against signatures and anomalies.

For our purposes there are at least two kinds of firewall:

- Network firewall. This limits access at the network level.

- Web application firewall. A web application firewall examines network traffic at the web application level to detect and/or limit damage. Its deeper inspection than typical network firewalls or IPSs can protect web applications/servers from web-based attacks that IPSs cannot prevent.

---

[15] Kryptowire also supports iOS, and it also includes some static analysis to provide context for the dynamic analysis. For our purposes, we are highlighting the forced path execution technique Kryptowire uses on Android applications with bytecode, as this provides a different analysis technique than is used by many other tools.

### b. Details

A firewall controls, monitors and analyzes incoming and outgoing traffic. Data packets are analyzed to determine whether they should be allowed in or not. Additionally, most firewalls today provide intrusion detection for both network and application layers. Application layer firewalls also typically provide virtual patching techniques along with a broader set of protocol analysis.

Network layer firewalls filter at the network packet level. They generally operate at a low level of the network protocol stack (TCP/IP, IP, UDP, etc.), controlling access to the network they are protecting by either allowing or not allowing packets to pass through. The controls used are based on a set of firewall rules that are defined/configured by the organization's security administration. Network firewalls may be either stateful or stateless. A stateful firewall maintains the context of the various active sessions and uses the state information to speed packet processing. Stateless firewalls have a generally higher throughput because they are not keeping state, but they have less information that they can use for decisions.

Application firewalls are differentiated from standard network firewalls because they can analyze higher-level protocols and data (e.g., File Transmission Protocol (FTP), Domain Name System (DNS), and HyperText Transfer Protocol (HTTP)). They may use deep packet inspection and integration with IDS/IPS. They may integrate with user authentication and authorization services to limit who has access to higher-level services. They may also provide virtual patching capabilities. Virtual patching is the quick development and short-term implementation of patch/configuration change to prevent an exploit from occurring if an intrusion or vulnerability is discovered. It is considered a temporary fix until proper mitigations are deployed.

For more information, see [McDowell 2009] and [Northrup 2013].

### c. Applicability

As with IDS/IPS, firewalls are useful in the operational environment because they can prevent some potential intrusions/breaches. This enables countering intrusions/ breaches directly, as well as possibly suggesting future changes to the system to make future attacks even more difficult to perform. Similarly, firewalls are useful in the development and sustainment environment, because they can sometimes prevent potential intrusions/breaches aimed at subverting the software in development or sustainment.

For software systems assurance, firewalls can be used to evaluate software by isolating the software being monitored. This may occur during more extensive processes such as monitored execution and penetration testing.

**d. Assessment**

Pros:

- The technology has already been widely adopted and used.

- It can provide simple access control limitations.

Cons:

- It can only provide partial isolation while still allowing data through.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:

  − Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.

- Recurring resource requirements:

  − Annual maintenance fees,

  − Usage effort.  This includes setting up the configuration, running the tool.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Firewall Type | URL |
|---|---|---|---|
| Juniper | Juniper Firewall Product line | Network firewall | http://www.juniper.net/techpubs/software/ive/esap/releasenotes/List_of_Products_Supported_By_ESAP_1.5.2.pdf |
| Cisco | Cisco ACE Web Application Firewall | Application firewall | http://www.cisco.com/en/US/products/ps9586/index.html |
| Sophos | Sophos UTM (formerly Astaro Security Gateway) | Client firewall | http://www.sophos.com/en-us/products/unified.aspx |
| Sonicwall | Dell™ SonicWALL™ Next-Generation Firewalls | Gateway Firewall | https://www.sonicwall.com/us/en/products/Next-Generation_Firewall.html |

## 46. Man-in-the-Middle Attack Tool

### a. Overview

A man-in-the-middle attack tool attempts to intercept and perform a man-in-the-middle attack on the application. This can be at the network level or in lower-level application communication protocols.

### b. Details

A man-in-the-middle attack can be conducted by an attacker who intercepts and establishes a connection between two victims while presenting himself as the two victims in their private connection. Attackers are likely to intercept messages going between the two victims and inject new ones. An attacker has to impersonate each endpoint in such a way that the victim is not able to differentiate between the attacker and the intended connection. Various countermeasures can be put in place for man-in-the-middle attacks, including strong encryption (communication encrypted to protect packets traversing network), public key infrastructures (mutual authentication), strong mutual authentication (secret keys, passwords), latency examination, and others, but these countermeasures can fail.

Attempting to perform a man-in-the-middle attack can help determine whether the countermeasures (if any) are adequate. These attacks can occur at many locations and protocols, including wireless or LAN based attacks, SSL/TLS, Secure Shell (SSH), and Android intents. Attempting to perform an attack is different than merely observing behavior (as with network sniffers); in particular, it may be possible to get systems to accept poorly-authenticated data using protocols or configurations they would not normally use (e.g., downgrading to a poor encryption algorithm).

### c. Applicability

This tool/techniques applies wherever a man-in-middle attack could occur.

### d. Assessment

Pros:

- Method of identifying man-in-the-middle characteristics or performing such attacks to test protection measures.

Cons:

- Often requires a technically adept analyst.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  - Small investment for licensing (many tools are open source),

  - Minimal uptime for implementing attack or detection mechanisms.

- Recurring resource requirements:

  - Annual maintenance fees are often minimal to none because of the number of open source tools available.

### f. Examples of Suppliers/Products

Many tools incorporate man-in-the-middle attacks as part of a larger set of attacks.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Rapid7 | Metasploit | Not applicable | http://www.metasploit.com/ |
| KARMA | Theta44 | various | http://www.theta44.org/karma/ |
| Ettercap | Ether cap | Linux | http://ettercap.github.io/ettercap/ |
| OXID.IT | Cain and Able | Windows environment | http://www.oxid.it/cain.html |
| Telerik | Fiddler | Windows environment | http://www.telerik.com/fiddler |
| Intrepidus Group | Mallory (TCP/UDP proxy) | Mobile environment – 802.11 http/https MIK attack tool | https://intrepidusgroup.com/insight/mallory/ |
| Monkey.org | DSNIFF (esp. sshmitm and webmitm) | Not applicable | http://monkey.org/~dugsong/dsniff/ |
| Thoughtcrime | SSLStrip/SSLSniff | various | http://www.thoughtcrime.org/software/sslstrip/ |

## 47. Debugger

### a. Overview

A debugger is a tool that enables observation and control of a program under execution. This can include the ability to execute the program step by step, and to observe internal states and results.

### b. Details

Debuggers allow programs to run under the monitoring and control of a human (directly, or via plug-ins to the debugger). Debuggers normally include (directly or indirectly) a disassembler and other tools to enable a human to determine the current state of a program.

Debuggers can provide some insight into how programs really work, and can detect certain signs of malicious behavior or unintentional vulnerabilities. However, the large effort and strong expertise required limits in practice what this approach can and cannot do with large, modern software. These tools present material at a very low level (especially if monitoring machine code). Additionally, debuggers can only present what happens with a particular set of inputs at particular points in time, not for all inputs. Thus, debugging use tends to be focused on spot checks or on very specific issues (e.g., for root cause analysis), instead of being used as a general-purpose tool to find arbitrary problems.

Malicious programs may include "anti-debugging" code that attempts to detect that it is running under the control of a debugger, and then change to benign behavior to evade detection. Debuggers can attempt to counter this, but this leads to an arms race between malicious code developers and debugger developers.

### c. Applicability

These tools require binary or bytecode. Source code improves their usability, but it can still be overwhelming to follow non-trivial programs using them.

For software systems assurance, debuggers can be used to help evaluate software by running the software under its monitoring. This may occur during more extensive processes such as monitored execution and penetration testing. The debugger may use plug-ins to help monitor suspicious or unusual behavior, and then enabled detailed review and analysis when suspicious events occur.

### d. Assessment

Pros:

- Debuggers can apply in cases where only the binary or bytecode is available.

Cons:

- Human analysis tends to be very costly and difficult to scale.

- Debugging only reports about a particular input set; different input sets would typically produce different results.

- Training is necessary.  Users must have deep and extensive knowledge of the lower-level constructs that are being analyzed.

### e.  Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Licensing fees,

    − Training, including costs for training, and the time of those receiving the training.  This includes in-depth knowledge of the underlying binary or bytecode format.

- Recurring resource requirements:

    − Annual maintenance fees,

    − Usage effort.  This can be substantial.

### f.  Examples of Suppliers/Products

Note that IDA Pro is not related to the Institute for Defense Analyses (IDA).  Also, IDA Pro is both a disassembler (static) and a debugger (dynamic).

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Hex-Rays | IDA Pro | https://www.hex-rays.com/index.shtml |

## 48.  Fault Injection

### a.  Overview

Fault injection techniques insert faults into software to enable better testing.  There are two main types:

- "Source code fault injection tools provide [mechanisms] through which source code can be instrumented to induce the code to follow control paths that would be otherwise difficult to test for."

- "Binary fault injection tools provide mechanisms through which safety- or security-related faults can be sent to the application while it is running….Unlike

source code fault injection, binary fault injection does not require knowledge of the application's source [code]." [BAH 2009]

## b. Details

The term "fault injection" is used with a variety of meanings in industry. Some sources appear to include any tool that sends input or other signals so that the tool can detect problems. For the purpose of this document, we use a narrower definition, focusing on tools designed to trigger error-handling code by intentionally creating errors. All tools/techniques in the dynamic or hybrid analysis category send data to detect problems, including the various application-specific vulnerability scanners (such as web application scanners), fuzz testing, and negative testing.

For our purposes, fault injection techniques intentionally insert faults. A key difference is whether this is done at the source code level or at the binary/bytecode (external) level.

Source code fault injection requires "a deep understanding of the software being tested. In return, testers achieve greater code coverage and a lower false positive rate than other testing methods. However, like static analysis tools, these tools require that the source code to the application be available." [BAH 2009]

Binary fault injection does not require knowledge of the application's source code, and "can successfully be performed with little specific training. However, additional analysis may be required to determine the best course of action for mitigating any defects identified by these tools." [BAH 2009]

Since fault injection simulates faults, it can be a useful technique for testing fault/error handling. However, it is less effective at detecting other vulnerability sources.

## c. Applicability

Source code is required for source code fault injection; binary/bytecode is required for binary fault injection.

## d. Assessment

Pros:

- Fault injection is helpful in finding problems in fault/error handling that might otherwise be missed.

- Binary fault injection is relatively easy to apply for use in detecting problems.

Cons:

- Source code fault injection requires strong knowledge of the application.

- It is primarily a way to detect problems in fault/error-handling, and not for many other problems.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  – Licensing fees.  Licensing schemes vary; some per node, others by lines of code analyzed.

  – Training.

- Recurring resource requirements:

  – Annual maintenance fees.

### f. Examples of Suppliers/Products

Note that many "fault injection" tools also perform other types of dynamic analysis (e.g., fuzzing), which are covered separately.

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Critical Software | csXCEPTION | http://asd.criticalsoftware.com/csxception/ |
| –A. Bertogli (open source) | libfiu | http://blitiri.com.ar/p/libfiu/ |

## 49. Logging Systems

### a. Overview

Logging systems records events, and their times, to provide an audit trail that can be used to understand software activity and diagnose problems.  The "syslog" service is an example. This information may be sent to a Security Information and Event Management (SIEM) system.

**b. Details**

Logging systems record events to provide an audit trail. They are used to understand the activity of software systems and to diagnose problems. Logs are essential to understanding the activities of complex systems. They can be used to combine log file entries from multiple sources. This approach, combined with various sophisticated analyses can result in the correlation and assessment of both related and unrelated events for anomalies.

Although an enterprise may be populated with a number of logging systems, they are usually a mechanism within various enterprise/system solutions such as operating systems, firewalls, intrusion detection systems, etc. Additionally, each vendor solution has logging mechanisms that are specific to their functionality and can monitor and track any number of things, including access, network traffic, change in file/filesystem, intrusions, and denial of service.

**c. Applicability**

Logging systems are a standard method of tracking events in both systems and a network. Common event mechanisms include syslog, which is found in an operating system and can track accesses to a system by tracking login mechanism or file system activities, including successful and unsuccessful attempts and unauthorized access attempts.

For software systems assurance, logging systems can be used to help evaluate software by maximizing what is logged, and then running the software while logging issues. This may occur during more extensive processes such as monitored execution and penetration testing. The logging system results may then be analyzed to gain insight into the software's behavior.

**d. Assessment**

Pros:

- These tools are helpful in quickly finding tracking events/activities.

- They are low cost or integrated into existing infrastructure components.

Cons:

- There may be low-level data collection requiring synthesis and correlation for more thorough analysis.

**e. Resource Requirements**

Resource requirements include:

- Integrating the software into the logging system, including appropriate selection of what to log.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Rsyslog project / Adiscon GmbH | rsyslog | http://www.rsyslog.com/ |

## 50. Security Information and Event Management

### a. Overview

Security Information and Event Management (SIEM). "SIEM technology provides real-time analysis of security alerts generated by network hardware and applications." [Dr. Dobbs 2007]

### b. Details

A SIEM delivers real-time analysis of security alerts from various network devices and software. SIEMs may be used as a way to log and analyze data (including network data and security alerts), and generate reports for analysis and compliance.

### c. Applicability

SIEMs are widely applicable, integrated, and used in the operational enterprise to track the activities in the network that may provide insight into the network operations for verification of expected behavior/events as well as anomalous behavior.

For software systems assurance, SIEMs can be used to help evaluate software by maximizing what is logged, enabling other monitoring systems, and then running the software while the SIEM gathers relevant data. This may occur during more extensive processes such as monitored execution and penetration testing. The results may then be analyzed to gain insight into the software's behavior.

### d. Assessment

Pros:

- SIEM functionality is fairly commoditized in the market.

Cons:

- Selecting a SIEM can be challenging, given the large number of available options and variation in their capabilities.

## e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
  - Licensing fees. Licensing schemes vary; some per node, others by lines of code analyzed.

- Recurring resource requirements:
  - Annual maintenance fees.
  - Usage effort. This includes setting up the configuration, running the tool, and reviewing the results to determine what vulnerabilities are applicable and their priority.

## f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| EMC | RSA Security Information & Event Manager | http://www.emc.com/security/rsa-security-information-event-management.htm |
| Enterasys | Enterasys Security Information & Event Manager | http://www.enterasys.com/products/advanced-security-apps/security-information-management/ |
| HP Enterprises | ArcSight | http://www8.hp.com/us/en/software-solutions/siem-arcsight/ |
| Intel Security | McAfee Enterprise Security Manager | http://www.mcafee.com/us/products/siem/index.aspx |
| Splunk | Splunk | http://www.splunk.com/view/enterprise-security-app/SP-CAAAE8Z |
| Tripwire | Security Information & Event Manager | http://www.tripwire.com/it-security-software/log-event-management/security-information-event-management-siem/ |

## 51. Test Coverage Analyzers

### a. Overview

Test coverage analyzers are tools that measure the degree to which a program has been tested (e.g., by a regression test suite). Common measures of test coverage include statement coverage (the percentage of program statements executed by at least one test) and branch coverage (the percentage program branch alternatives executed by at least one test). Areas that have not been tested can then be examined, e.g., to determine whether more tests should be created or whether that code is unwanted.

### b. Details

Software can be run through a large series of tests, but even a large set of tests may completely fail to test significant portions of the software.

A "test coverage criterion" defines a criterion for measuring completeness of a test. As noted above, common measures are statement coverage (the percentage of software statements executed by at least one test in the test suite) and branch coverage (the percentage of branch alternatives executed by at least one test in the test suite). Achieving 100% coverage can be expensive for some software, but it may be possible to examine those parts not covered to determine *why* they were not covered by a test, and in particular, determine whether or not these untested regions are a problem.

This approach can, in particular, identify some Trojan horses. If the software includes malicious logic that is not triggered by any test, then the Trojan horse will be among the untested portions that can be examined later. Note, however, that malicious developers can take measures to evade detection (e.g., by implementing built-in interpreters and hiding the malicious code in interpreted code).

Some code coverage tools require debug symbol information, or require that the code be recompiled to measure code coverage. These can be significant limitations, depending on the information available.

### c. Applicability

These approaches are more difficult (and expensive) to do without source code. These approaches require the use of a test suite; developers typically already have a regression test suite, but end-users typically will not.

Thus, from a software assurance perspective, potential users may want to negotiate with the supplier to obtain a regression test suite or even the source code if they wish to apply this technique.

**d. Assessment**

Pros:

- This approach can quickly identify portions of software that are untested; untested software is particularly likely to be in error, and this includes the potential for vulnerabilities.

- It can detect certain kinds of malicious code.

Cons:

- It can be expensive, especially if the regression test suite has poor coverage. This can happen if many situations are difficult to trigger (and thus difficult to test).

- Malicious developers can evade detection if they work to do so.

**e. Resource Requirements**

Resource requirements include:

- Initial resource requirements:
  - Obtain test coverage measurement tools,
  - Examine each untested area, to document it or to create tests to cover it.

**f. Examples of Suppliers/Products**

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Languages Supported | URL |
|---|---|---|---|
| GNU | gcov | C, C++, Ada, Fortran | |
| EclEmma team | JaCoCo | Java bytecode | http://www.eclemma.org/jacoco/ |
| Atlassian | Clover | Java | http://www.atlassian.com/software/clover/overview |

## 52. Hardening Tools/Scripts

### a. Overview

Hardening tools and scripts modify software configurations to counter or mitigate attacks, or to comply with policy. In the process, they may detect weaknesses or vulnerabilities in the software being configured.

### b. Details

Hardening tools and scripts can counter or mitigate attacks by changing configurations. For example, they can disable some connections (reducing the attack surface), disable some encryption algorithms (which may be weak against attack), and increase difficulty of attack (e.g., by requiring stronger certificates and passwords). Such tools often do not simply change the configurations, but report current values and identify proposed changes. Hardening tools and scripts are a normal part of improving resistance to attack.

DISA Security Technical Implementation Guides (STIGs), http://iase.disa.mil/stigs/, provide hardening guidance for many products, as do NSA Guides. Historically these were often long checklists of manual configuration steps to take during or after installation, but significant progress has been made on automating these processes.

The Security Content Automation Protocol (SCAP) is a synthesis of specifications to support security automation. The SCAP language specifications are Extensible Configuration Checklist Description Format (XCCDF), Open Vulnerability and Assessment Language (OVAL®), and Open Checklist Interactive Language (OCIL™). In particular, an "authenticated configuration scanner" is a tool that can be used to implement configuration hardening. Covering SCAP in detail is beyond the scope of this document; for more information on SCAP see http://scap.nist.gov/.

### c. Applicability

Hardening tools and scripts are typically created for widely-used OTS components.

As an *analysis* or *evaluation* tool, this approach applies primarily to systems that include widely-used OTS components that can be accessed by these tools. Many applications include, directly or indirectly, components that can be hardened using hardening tools and scripts. These components range from cryptographic libraries to entire operating systems. Hardening tools and scripts can often report the current settings, particularly if they are not the recommended settings; such deviations from recommended settings may indicate a lack of concern about security in the overall product. In addition, the hardening tools and scripts can be used to harden these embedded products, followed by testing of the system that includes them; if the product

cannot work with hardened settings, those changes can be examined to determine whether this indicates a lack of concern about security. Finally, in some cases a component needs to be hardened to meet some policy; the process of trying to developing hardening scripts to meet that policy is likely to reveal problems that may suggest the product should or should not be used at all.

### d. Assessment

Pros:

- Hardening tools and scripts can apply to many larger systems.

Cons:

- They are often indirect indicators of concern for unintentional vulnerabilities, not direct indicators.

- Understanding the implications of the results requires an understanding of the hardened components and the larger system.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Obtaining tools,
    - Training.

- Recurring resource requirements:
    - Annual maintenance fees,
    - Examining the results to determine their impact and potential implications.

### f. Examples of Suppliers/Products

Hardening is a technique, not a tool, and many tools can be used for hardening. A list of products that have been validated by NIST as conforming to the Security Content Automation Protocol (SCAP) and its component standards is available at http://nvd.nist.gov/scapproducts.cfm.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Rapid7 | Nexpose | http://www.rapid7.com/products/nexpose/ |
| Microsoft | SCAP Extensions for Microsoft System Center Configuration Manager 3.0 | https://www.microsoft.com/ |
| Tenable | SecurityCenter | http://www.tenable.com/products/securitycenter |
| OpenSCAP project / Red Hat | OpenSCAP | https://www.open-scap.org/ |

## 53. Execute and Compare with Application Manifest

### a. Overview

An execute and compare with application manifest tool runs an application with a variety of inputs to determine the permissions it tries to use, and compare that with the application permission manifest.  In practice it may be guided by other information.

### b. Details

As with tools that use dynamic analysis, it can report only what occurs given the inputs used.  Static analysis of code can help increase the coverage and focus of the dynamic analysis, depending on the depth of the static analysis used.

### c. Applicability

This type of tool/technique requires that there be a permission manifest (e.g., such as in the Android operating system).

### d. Assessment

Pros:

- Simple and easy to apply.

Cons:

- They do not directly find vulnerabilities (necessarily), but instead warn of potentially high-risk sets of permissions.

- Can only report differences based on the set of input data used.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

    − Obtaining tools,

    − Training.

- Recurring resource requirements:

    − Annual maintenance fees,

    − Examining the results to determine their impact and potential implications.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | URL |
|---|---|---|
| Open Source | Permlyzer | http://www.cse.psu.edu/~szhu/ papers/permlyzer.pdf |
| Kryptowire | Kryptowire (Android edition) | http://www.kryptowire.com/ |

## 54. Track Sensitive Data

### a. Overview

The track sensitive data tool/technique statically identifies sensitive data or data sources (e.g., due to privacy concerns or confidentiality requirements) and then dynamically executes the application using the data, tracking it to observe/detect exfiltration attempts or misuse of the data.

### b. Details

This approach tracks data as it flows through the system, often down to the variable assignment level.  Historically this approach has been challenged by the overhead it imposes, although progress has been made on this front.  The approach requires that sensors be appropriately placed for observation of data through flows.

### c.  Applicability

This technique is well-suited for smaller footprint applications, such as mobile applications.

### d.  Assessment

Pros:

- Can detect potential exfiltration of sensitive data, even through multiple complex layers of functionality,

- Can be used to detect unacceptable data sharing between applications.

Cons:

- Only detects problems when they dynamically arise,

- Requires a strong understanding of what is sensitive (and thus requires protection) in a given context.

### e.  Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Investment for licensing of various tools,
    - SME expertise in systems and security analyses.

- Recurring resource requirements:
    - Annual maintenance fees.

### f.  Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
|  | TaintDroid | Android | http://appanalysis.org/ |
| Berkeley | TaintEraser |  | http://appanalysis.org/privacyscope/osr_tainteraser.pdf |

## 55. Coverage-guided Fuzz Tester

### a. Overview

A coverage-guided fuzz tester is a fuzz tester that uses code coverage information to determine new inputs to test.

### b. Details

As noted in the section on fuzz testers (in subsection C.38), **fuzz testing** or **fuzzing** is an automated black box testing technique for software that involves providing invalid, unexpected, or random data to the inputs of a computer program. See that section for more about fuzzing in general.

A coverage-guided fuzz tester uses information about code coverage information to help guide it to new inputs for testing. It may monitor coverage of statements, branches, or data flows. This coverage information may include the number of times something has occurred, instead of simply noting whether something has occurred at all. This additional information can, in some cases, enable a fuzzer to detect and check paths that many other fuzzers cannot detect [Zalewski2014].

A coverage-guided fuzz tester is a hybrid tool, not a strictly dynamic tool, since these tools take advantage of static code information.

### c. Applicability

Coverage-guided fuzz testing is in theory a widely applicable testing technique for software. However, it does require the ability to monitor the paths of a program being executed (something that traditional fuzzing typically does not do). At the time of this writing coverage-guided tools do not handle network protocols well, although further research and development may change this.

### d. Assessment

Pros:

- Fast ramp-up for use,

- Fast method of identifying some critical issues in software systems,

- Extremely cost effective approach.

Cons:

- Not a comprehensive testing approach,

- Only certain types of security flaws would typically be discovered by fuzz testing,

- Typically has diminishing returns.

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Small investment for licensing (fuzz testers are often inexpensive compared to other analysis tools).

- Recurring resource requirements:

  − Annual maintenance fees are often minimal to none.

### f. Examples of Suppliers/Products

Many programs implement fuzz testing. In addition, some interviewees stated that they found it just as effective to write specialized fuzzing programs, since they are fairly easy to write and specialized fuzzers can target properties of specific applications and platforms.

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|----------|-------------|----------------------------|-----|
| Google | American Fuzzy Lop (AFL) | POSIX (including Linux) | http://lcamtuf.coredump.cx/afl/ |
| LLVM | LibFuzzer | POSIX C/C++ | http://llvm.org/docs/LibFuzzer.html |

## 56. Probe-based Attack with Tracked Flow

### a. Overview

The probe-based attack with tracked flow tool/technique observes normal behavior while tracking data and control flows within the program (possibly through several tiers), sends probing inputs to determine patterns of behavior that might indicate a potential vulnerability, then based on these patterns, performs simulated attacks to identify actual vulnerabilities.

## b. Details

These tools combine dynamic analysis (because the code is executed) with static analysis (because the statically defined control and data flows of the program are monitored for behavioral changes). This approach has similarities to coverage-guided fuzz testing; however, it has information on what is "normal" behavior (and can exploit this difference). This approach also has similarities to web application scanners, and some may even combine them in one category, but unlike web application scanners (as defined in this paper), these tools additionally use static analysis approaches to track data and control flow.

## c. Applicability

This approach may be especially helpful when the software can be viewed as repeatedly taking input and then responding to it, e.g., a typical web application.

## d. Assessment

Pros:

- Fewer false positives, since reports are based on actual execution of simulated attacks.

Cons:

- Significant risk of false negatives; these types of tools can report a potential problem only if that path is executed and can create an exploit. This risk can be reduced by improving its automated test suite coverage.

## e. Resource Requirements

Resource requirements include:

- Initial resource requirements:
    - Investment for licensing of various tools,
    - SME expertise in systems and security analyses.
- Recurring resource requirements:
    - Annual maintenance fees.

## f. Examples of Suppliers/Products

The following are examples of suppliers and their products. These examples are provided to help readers understand this tool/technology type. This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|---|---|---|---|
| Synopsys | Seeker[16] | Java JVM, .NET framework, PHP, others | http://www.coverity.com/products/seeker/ |

## 57. Track Data and Control Flow

### a. Overview

The track data and control flow tool/technique tracks data and control flows from inputs and other data sources to data sinks, and reports when rules (predefined or user defined) are triggered indicating a potential vulnerability.

### b. Details

These tools combine dynamic analysis (because the code is executed) with static analysis (because the statically defined control and data flows of the program are monitored). For example, such a tool may determine that a data flows from an untrusted source (e.g., an untrusted user input) to a sensitive sink (e.g., a SQL command execution engine) without going through a validator or sanitizer. These tools do not necessarily need to receive attack input, since they can simply monitor taint as it flows through a system, but they do have to execute the potential path to report it.

### c. Applicability

These approaches are especially helpful when the software can be viewed as repeatedly taking input and then responding to it, e.g., a typical web application.

### d. Assessment

Pros:

- Fewer false positives, since reports are based on actual execution.

Cons:

- Significant risk of false negatives; these types of tools can only report a potential problem if that path is executed. This risk can be reduced by improving its automated test suite coverage and making it easier to scan all paths.

---

[16] Note that Seeker includes the ability to monitor and correlate information across tiers (e.g., web application server, back-end/servlet, and DBMS), which can aid its analysis (e.g., it can track code inside the DBMS that is executed as stored procedures and correlate that with the corresponding request).

### e. Resource Requirements

Resource requirements include:

- Initial resource requirements:

  − Investment for licensing of various tools,

  − SME expertise in systems and security analyses.

- Recurring resource requirements:

  − Annual maintenance fees.

### f. Examples of Suppliers/Products

The following are examples of suppliers and their products.  These examples are provided to help readers understand this tool/technology type.  This list is illustrative, and no endorsement is implied.

| Supplier | Product Name | Language or Application Type | URL |
|----------|--------------|-----------------------------|-----|
| Contrast Security | Contrast | JVM, .NET, Node.js, ColdFusion | https://www.contrastsecurity.com/ |

Note that Contrast also includes capabilities that are separately listed under origin analyzer, web application vulnerability scanner, configuration checker, and bytecode weakness analyzer.  See those tool/technique types for more information.
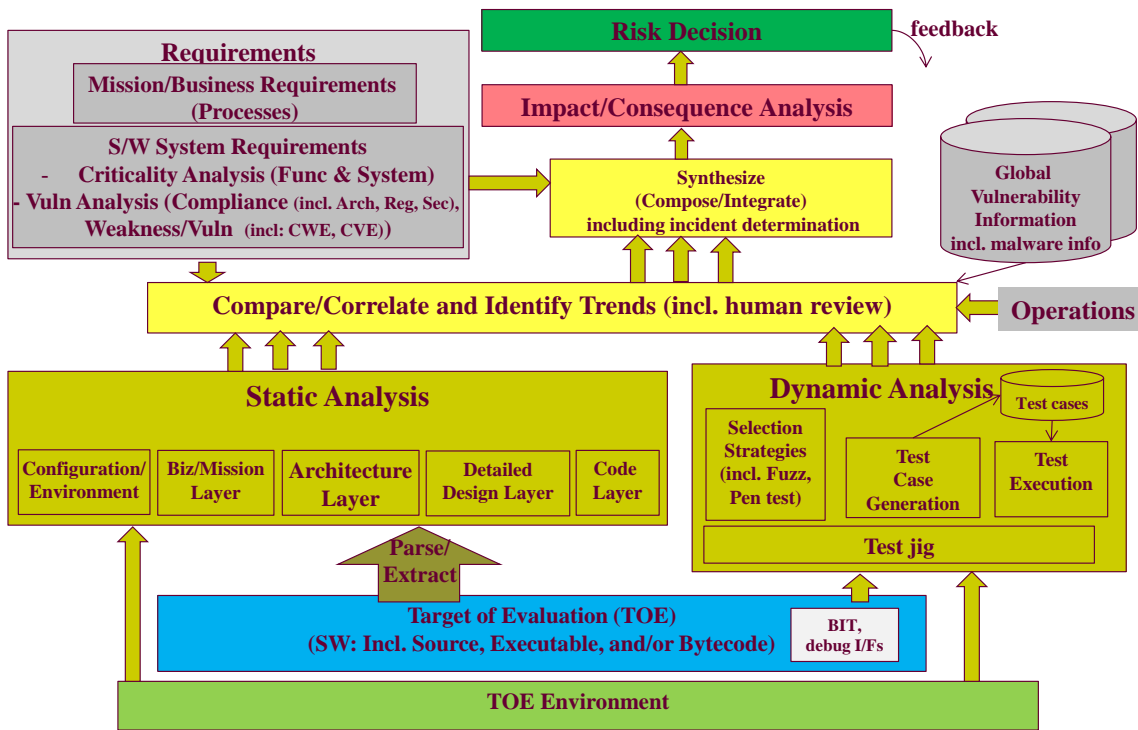
# Appendix D.
# Detailed Compositional Views

Currently tools and techniques for assurance are often discretely applied within specific software development lifecycle (SDLC) processes. There is often little integration, correlation, and syntheses of analysis tools and results throughout the lifecycle. We believe that analysis should instead be integrated into the SDLC, from development (including developing requirements, design, implementation, and test), operations, sustainment, and disposal.

Although some organizations already integrate analysis throughout the lifecycle, this can be especially complex in the DoD. The DoD is highly distributed and large-scale, and it has widely varying environments (from "office to desert"). In addition, the DoD uses many OTS components, making it more difficult for the DoD to "build in" trust.

We encourage program managers to view analysis as input to risk decisions holistically, throughout the lifecycle. The following two figures show how analysis results could be integrated throughout the lifecycle in a continuous manner. The first shows analysis during development and sustainment, where analysis results are compared and correlated, synthesized, examined to determine their impact and consequence, leading to a risk decision. Risk decisions may change any part, creating a feedback loop throughout the lifecycle. Once a system is in operations, information derived from operation should also be used as input.

Policymakers should also consider an enterprise view. They should work to gather information from the various systems (as noted above) to determine overall enterprise trends. Given that information, they should then determine the key gaps in the analysis techniques that make it difficult to deliver software with the requisite software assurance and acceptable supply chain risk for its missions.

# Analysis During Development and Sustainment

**Risk Decision** — feedback

**Impact/Consequence Analysis**

**Requirements**
- Mission/Business Requirements (Processes)
- S/W System Requirements
  - Criticality Analysis (Func & System)
  - Vuln Analysis (Compliance (incl. Arch, Reg, Sec), Weakness/Vuln (incl: CWE, CVE))

**Synthesize (Compose/Integrate) including incident determination**

**Global Vulnerability Information incl. malware info**

**Compare/Correlate and Identify Trends (incl. human review)** — **Operations**

**Static Analysis**
- Configuration/ Environment
- Biz/Mission Layer
- Architecture Layer
- Detailed Design Layer
- Code Layer

**Parse/ Extract**

**Dynamic Analysis**
- Selection Strategies (incl. Fuzz, Pen test)
- Test Case Generation
- Test Execution
- Test cases
- Test jig

**Target of Evaluation (TOE) (SW: Incl. Source, Executable, and/or Bytecode)** — BIT, debug I/Fs

**TOE Environment**

**Analysis During Development and Sustainment**

---

# Analysis During Operations

**Results (to compare & correlate)**

**Both pre & post deployment (In pre-deployment, to-be testing in staged op. env.)**

**Abstracted Data/Information**
- Network Architecture (incl. hosts, interconnects) w/ TOE
- Detailed Design Layer (D/ B tables, data flow over interconnections)
- Operational Configuration incl. Hardening (TOE & its env, incl. network)
- Access Control Constraints (TOE & its env)
- Problem Reports (bugs, features, feedback, ..)
- Patch Mgmt incl. firmware patches (availability, expected impact, patch status)
- N/W and Host Scan/Test Results (incl. Security)
- Upgrade Mgmt incl. firmware upgrade
- IDS/IPS/ incident or other Intrusion Alerts & Forensics
- Audit Information (logs - what actually happened)

**Extract**

**Target of Evaluation (TOE): Operational Representation**

Filter shaped by TOE

**Feedback (e.g., users, admins, external orgs)**

**TOE Environment Representation (network infrastructure where TOE resides)**

**Analysis During Operations**

# Appendix E.
# Software State-of-the-Art Resources (SOAR) Matrix

For the contents of Appendix E, refer to the separate electronic file provided to the sponsor containing the Software State-of-the-Art Resources (SOAR) Matrix.

# Appendix F.
# Mobile Environment

This appendix focuses on software assurance for mobile environments. Mobile computing has been defined as "being able to use a computing device even when being mobile and therefore changing location. Portability is one aspect of mobile computing." [MobileMAN]

The DoD Mobile Device Strategy [2012] defines a mobile device as "a handheld computing device with a display screen that allows for user input (e.g., touch screen, keyboard). When connected to a network, it enables the sharing of information in formats specially designed to maximize the use of information given device limitations (e.g., screen size, computing power). Mobile devices provide the conveniences of conventional desktop or laptop computers in a more portable package. Popular form factors for mobile devices are smart phones and tablets." For our purposes, we focus on smartphones and tablets, such as those running Android or iOS operating systems, as mobile devices. We do not further distinguish between smartphones and tablets since, although they are different in size, they are otherwise similar. In many cases they run the same operating systems and applications, and while some tablets cannot connect to cell towers, others have this capability built in.

We do distinguish mobile devices from basic cell phones and laptops. Basic cell phones have limited functionality, such as making phone calls, texting, and searching the web, but lack a rich application framework capable of supporting a broad set of applications. Laptops can run many applications, but are less portable.

The following sections provide an overview of mobile components for the enterprise, aspects of mobile computing that are differentiators or especially relevant compared to other computing environments, and changes that have been made to the first draft of this paper specifically to discuss mobile computing issues.

## 1. Mobile Components for the Enterprise

Mobile systems are more than just the mobile hardware carried by users. For example, the DISA STIGs divide mobile systems further by defining four mobility security requirement guides (SRG) that "must be considered together when implementing an enterprise mobility solution within DoD." [DISA STIG MDM]:

- Mobile operating system (OS) SRG "addresses security for the operating system installed on mobile devices."

- Mobile Device Management (MDM) SRG "addresses centralized management of mobile operating systems and applications. The MDM SRG also covers aspects of device integrity verification and enterprise email."

- Mobile Applications SRG "addresses the security of applications that run on mobile OS."

- Mobile Policy SRG "addresses management, operational, personnel, and physical security controls related to mobile devices."

Some organizations support or use a "bring your own device" (BYOD) policy, in which users own the mobile device, but the enterprise data continues to belong to the enterprise. DoD policy does not approve of BYOD for DoD purposes [DoD CIO].

Some enterprises do not use an MDM; in that case, they may choose to use a mobile application management (MAM) approach instead. MDMs enable an enterprise's information technology (IT services) to "secure access to the device by requiring the use of a passcode and keep sensitive data out of the wrong hands by remotely wiping a lost or stolen device. Other basic features of MDM tools include the ability to enforce policies, track inventory and perform real-time monitoring and reporting." For some organizations, this may be too heavy-handed, particularly those with a BYOD policy. MAM approaches focus on managing only specific applications, giving enterprises the ability to "manage and secure only those [applications] that were specifically developed to work with a particular MAM [system, such as deleting corporate email without deleting personal photos]." [Steele 2013] [Madden]

A particular mobile device contains a large stack of software, including an operating system, various middleware that provides common services, in some cases an MDM agent, and a set of applications. In practice, many mobile device applications depend on external services (often running in a cloud), and these services may have access to the same data the application has.

Many enterprises also use additional software specifically to help separate business/enterprise data from personal data while supporting collaboration, typically in combination with an MDM or MAM. Examples of such software include "Good for Enterprise" from Good Technologies and "Secure Workspace" from Blackberry Ltd, both supporting Android and iOS environments. These typically provide services such as business/enterprise email, enterprise infrastructure web browsing, attachment viewing, and document creation.

It is important to note that there are differences in threats and use cases for traditional consumers, enterprises, and users in a tactical environment. For example, in a

tactical environment, communication is unreliable and slow, and an inability to use some data (including due to interruptions) could result in death.

Many documents exist that discuss securing mobile devices in the enterprise, including those focusing on federal government needs. These include Government Accountability Office (GAO) report GAO-12-757 [GAO-12-757], NIST SP 800-124rev1 [NIST 800-124], and the already-noted DoD STIGs. At the time of this writing, NIST is in the process of documenting technical considerations for vetting applications for Android mobile devices [NIST 800-163], although this is not publicly available. The Federal CIO Council has published a number of documents (see [Fed CIO Council 2013], [Fed CIO Mobile Security 2013], [Fed CIO Mobile 2013], [Fed CIO Arch 2013], and [Fed CIO Use Case 2013]). The European Union has also published relevant material [ENISA[17]].

The DoD Mobile Device Strategy [DoD Strategy 2012] guides overall DoD strategy for mobile devices, and has been followed up by a Department of Defense Commercial Mobile Device Implementation Plan [DoD Implementation]. The implementation plan includes plans to establish an enterprise mobile application store (MAS) capability that operates in conjunction with an MDM system. DISA has established, in response, the "DoD Mobility Program Management Office (PMO)" [DISA DoD Mobility PMO].

## 2. Mobile Computing Differentiators and Issues

Mobile devices are fundamentally computers designed to move with users. Thus, they are subject to many of the same issues as laptops and desktops. This is particularly true for laptops; both laptops and mobile devices can typically use Wifi networks, both can access internal networks (if configured to do so), and both directly interact with users.

However, some issues are specific, or often more important, on mobile devices. Our interviewees and other sources identified several issues that we have merged into larger groupings as follows:

1. Expectations and behavior by users and enterprises are different with mobile devices.

    a. *Consumerization*. Developers of mobile devices and operating systems tend to focus on the consumer market (a trend that is sometimes called "consumerization"). Features for enterprises have been added, but they are often not the driver for mobile device development.

---

[17] European Network and Information Security Agency

b. *Expected ease-of-use*.  Users and enterprises expect that mobile devices can be easily used for basic functions without any user training.  This expectation is not always met, but it still influences which applications users and enterprises select.  As a result, usability is very important.

c. *Tactical, enterprise, and general consumer users differ in their expected use and risk tolerance.*  In particular, in the tactical environment, communication is limited and unreliable, so mobile applications must sustain data within the device, e.g., a map application cannot rely on a remote tile server always being available in a tactical environment.  Also, tactical users do not want to flip between different applications that use a map; they need to be able to see one map and decide what information is important to display.   Finally, in a tactical setting, interruptions can be lethal.  Also, in general consumer use, risk tolerance tends to be higher than in enterprise and tactical environments.

d. *Bring your own device (BYOD) issues*.  Enterprises must decide whether they will use a BYOD policy, and if so, what its details will be.  A BYOD policy involves new issues not typically addressed previously by enterprises.  For example, if a smartphone is subverted, who is responsible, the user or the enterprise?  Fundamentally, the enterprise owns the business data, but under a BYOD policy the device may be owned by the user, creating a potential for conflict.  Some products, such as Mobile Device Managers (MDM) and GOOD, try to provide enterprise control of a mobile device.  BYOD creates some additional risks, which may be acceptable in some environments and not acceptable in others.  Note that the DoD STIGs reject BYOD and establish mobile devices as government-issued devices, along with DoD repositories of approved applications.

e. *Application installation is expected to be safe*.  There is a widespread expectation by users that installing a mobile application is relatively safe.  Mobile device suppliers work hard to limit application privileges, provide mechanisms to isolate applications from each other unless the applications permit it, and vet applications.  However, these are necessarily imperfect steps.  In part, this assumption may be because mobile phones users continue to think of their smartphones primarily as communication devices (e.g., for phone calls and text messages).  People often do not understand the additional risks that are created by installing additional applications.  Thus, while malicious applications can be a problem for any system, the assumption that applications are safe can work against the user.

f. *Users expect that they will not wait long to install an application*.  This expectation can limit the amount of independent vetting (e.g., of privileges and behavior) that can be done on third-party applications before they are

allowed to be installed on a device.  There is significant market pressure for a rapid application analysis capability on the order of less than an hour.

g.  *Users often grant application privilege requests without thought*.  In practice, users often just permit whatever the application requests.  One experimenter gave a set of students a tic-tac-toe game that requested many unnecessary privileges (e.g., to take audio and video), and yet only one user questioned it, and all students agreed to install it.

2.  Mobile devices have a big externally-accessible attack surface that is even more accessible to external attackers than laptops.

a.  *Always on*.  Mobile devices are typically operational at almost all times.  This is in contrast to laptops, which are often not active when being moved.  As a result, there are more opportunities over time for an attacker to subvert a mobile device.

b.  *Always communicating*.  Mobile devices are typically always communicating with external systems.  Smartphones typically have access to 3G/4G networks as well as Wifi, Bluetooth, and sometimes near field communications; as a result, their network communication capability is immediately usable almost anytime and anywhere.  This makes it easy to send information out without user intent or knowledge, provides a beacon for anyone to find that device or user, and creates a constantly available network attack surface of the device.  The device can be attacked at any time through the communication layer, even when it is in the owner's physical possession.

c.  *Always with the user*.  Mobile devices are nearly always with their user, giving the devices ample opportunity to collect data about the user for potential exploitation.  Note that typically mobile devices have more sensors and actuators (as discussed below) than laptops or desktops, creating a potentially dangerous combination.

d.  *Second processor and operating system for external communication*.  Devices that support mobile communications capability (e.g., 3G or LTE) typically use a separate baseband processor running a separate real-time operating system (RTOS) and programs that manage everything related to the radio and often other capabilities as well (e.g., GPS and USB).  Most users are unaware of this processor, RTOS, and applications, in contrast to the user-visible operating environment (e.g., Android or iOS).  The baseband processors and their associated software are typically poorly understood, poorly documented, and not externally peer reviewed.  In addition, the baseband processor typically trusts whatever data it receives from a base

station (e.g., in a cell tower). Remarkably, the baseband processor is usually the master processor, whereas the application processor (which runs the mobile operating system) is the slave. Thus mobile devices are exposed to over-the-air attacks that may enable total control of the device, yet these attacks may be poorly countered or mitigated [Holwerda 2013].

Note that some non-mobile computers also have separate processors and/or communication channels that can override the operating system. For example, Intel Active Management Technology (AMT) is hardware and firmware technology for remote out-of-band management of personal computers; this provides remote functionality even if the computer is powered off (as long as the power is provided), and can perform functions such as remotely powering up, power recycling, changing BIOS settings, and rebooting to a different operating system.

3. Mobile devices have a big internally accessible attack surface, which differs from user expectations that applications are isolated.

   *Applications can exploit inter-application communication mechanisms.* Applications can communicate directly or indirectly with other applications or services (including network services). As a result, it is sometimes easy for one application to get around restrictions placed on it by communicating with another application or service without user knowledge. This is counterintuitive to the user expectation that installing different applications is safe.

4. Mobile devices have additional assets that require protection.

   a. *Mobile devices have more sensors and actuators.* Mobile devices typically include GPS receivers (for location), camera, microphone, accelerometers, speakers, vibrators, video displays, and so on. Other devices (e.g., laptops) may have some of these, but typically not as many or as accessible. These sensors and actuators, coupled with being always on and always with the user, means that mobile devices often have significant access to private information (e.g., where someone goes and how long), and enable the synthesis of private and public information. As a result, if subverted they can become privacy-eliminating monitoring devices, enabling stalking and other dangerous behaviors.

   b. *Mobile devices include significant amounts of personal private data.* Mobile devices store information such as contacts and calendars, as well as overall user behavior. This, combined with their always being with the user as well as having more sensors (including location information), concentrates a great deal of private information. This private information is lucrative to some.

c. *Mobile devices encourage concentration of both personal and business data into one device.* People often do not want to carry two phones, and having two phones makes it difficult to merge data (e.g., to create a common calendar with both personal and business commitments). This merging creates a potential conflict for enterprise management:

   i. An enterprise decision to "erase the phone" might erase personal data as well, leading to hesitation to erase the phone.

   ii. Mobile devices encourage data storage and backups on their commercial cloud services. However, enterprises may not want their data stored there, since these other services may not protect or may even exploit that enterprise data.

   iii. Users often connect to external services (Google, Facebook, banking, cloud services, etc.), which then provide an avenue for malicious access and software to enter the mobile device and extract, modify, erase, or make unavailable data and services (both personal and enterprise).

5. Malicious applications can create behavior undesired by the user or enterprise. Malicious applications are a problem for any computing environment, but some aspects of the mobile environment create distinctive opportunities for exploitation.

   a. *Grayware.* A significant threat is grayware, that is, applications that are vaguely legitimate but that push the boundaries or go beyond what the user or enterprise may be comfortable with. For example, they may collect and redistribute user data for monetization or user profiling. One tool supplier said that their customers were really worried "about risky behavior and privacy issues… [they're] not really looking for vulnerabilities like on other platforms, [they're] looking for intended undesired behavior."

   b. *Application collusion.* Many mobile users are not aware that applications may collude; yet if applications collude, the impact can be significant. Different applications may have limited privileges that individually make sense, but together can work around the limitations imposed by the operating system. There are two types of application collusion: direct collusion (e.g., sharing files or directly sending messages to each other) and covert channels.

6. There is a lack of transparency for mobile users and software developers.

   a. *Little information provided to users and enterprise.* Relatively little insight is provided to users and enterprises that reveals what information is being shared or transmitted between applications, or between applications and the network. The information sharing could be either an unintentional

information leak or intentional malicious sharing not desired by the user. There is some information, e.g., in an application manifest or when an application requests a large functionality, but this is often not enough, and users often do not have enough knowledge to take action on it. This lack of insight makes it difficult to have a "rich feedback loop" (as one interviewee put it) where users can object to undesired functionality for security reasons.

b. *Little information provided to developers.* Third-party libraries are typically used to develop applications, but in many cases the developers do not know what the libraries really do, and the libraries are available only in executable form. As a result, third-party libraries may choose to do malicious things (such as share personal information) in ways that neither the end user nor application developer nor any of their organizations are aware of. Some third-party libraries intentionally violate privacy so that their makers can monetize their library. This creates undesirable incentives, since the library developers are paid to do this. Even enterprises often don't know exactly what activities are performed by the applications they have developed because of these third-party libraries.

7. Mobile devices have key characteristics in hardware, operating system/middleware, and application development that are unique or especially important. These characteristics create benefits and limitations specific to the mobile environment.

a. Hardware

i. *Small form factor.* Mobile devices' small form factor leads to different threats in different deployment scenarios. They are small and thus easy to lose, hand over, lose custody of, or have stolen (temporarily or permanently). For example, an adversary could acquire it in combat.

ii. *Limited local computing resources.* Mobile devices typically have far more limited resources in terms of memory, storage, and processing as compared to laptops and desktops. This implies that software must often be developed specifically for the device. See the discussion below on limited computing resources.

iii. *Limited electrical power.* Mobile devices have limited electrical power, and are more likely to be used without being plugged into power. Applications can unintentionally or intentionally drain battery power. This can result in a denial-of-service that isn't usually considered the same way on laptops or other general-purpose computers. This increases the opportunity for "juice jacking," where a charger is offered

that clandestinely attacks the device [Krebs] (although there are devices that can counter this attack).

b. OS & middleware

    i. *Applications must be given permission to perform some actions.* On Android, many of these privileges are granted at install time through a static manifest that comes with the application. On iOS, these privileges are primarily obtained through one-time requests by the application at runtime after installation. This is in contrast to traditional Windows or Unix-like desktops (including MacOS), where, typically, applications run with full user permissions and have access to the data developed by other applications.

    ii. *Enterprises can control what applications are used.* Enterprises can use MDMs, application stores, and other mechanisms to limit which applications and associated privileges can be used by mobile devices approved for their enterprise. This "whitelisting" approach for applications is like a "walled garden"; this was unacceptable to many PC users, but it seems to be acceptable to many mobile users.

    iii. *Application development is more like extending a framework.* Mobile applications are built into a rich ecosystem that lets them work with other applications and the underlying platform. Applications are not really standalone; rather, they are an extension of an existing framework combined with a large set of third-party libraries. This is true to some extent for all modern systems, including servers, desktops, and laptops (e.g., .NET framework and J2EE), but in mobile applications it is typically impractical to avoid them.

c. Applications

    i. *Mobile application software is event-driven.* While many non-mobile programs are event-driven, all mobile applications are inherently event-driven and interconnected to other components. This makes them more challenging to analyze with static tools. The event-driven nature makes some kinds of dynamic analysis easier during testing, but performance limitations greatly limit the types of dynamic analysis and monitoring that can be done operationally.

    ii. *Mobile applications typically undergo rapid change.* There are millions of mobile applications, and both the initial development and update lifecycle are often relatively short. Among actively updated applications, "the average update cycle for apps with at least 10 versions

is over two months on iOS but just a month and a half on Android. Windows Phone is the fastest-updating, at just over a month." [Koetsier] A multi-month analysis process is incompatible with these update cycle times; analysis must fit within these compressed mobile computing times. A primary market pressure is for rapid application analysis capability on the order of less than an hour. Although there is still a place for in-depth analysis, the challenge is to perform useful analysis within limited time as dictated by market pressures.

8. Application stores

   a. *Application stores can help improve security*. An application store is a repository for submitting and downloading applications. Application stores set minimum requirements for applications, perform evaluations, host user reviews, and ensure that application developers cannot make a malicious variant for just particular users. These can make malware distribution more difficult and simplifies offline analysis.

   b. *Application stores are no panacea*. User reviews are typically for functionality, not for security. Thus, a threat agent with resources can deliver a well-functioning application that can also perform malicious activities. Application stores can track what users are installing and provide malicious variants for specific users. Whether or not this is significant depends on the level of trust granted the application store. Mobile devices running Apple iOS normally only allow the Apple application store to be used. It is possible to use other application stores on Android devices (depending on the device configuration).

9. The mobile environment is newer and rapidly evolving, potentially opening significant vulnerabilities.

   *New platforms may provide new vulnerabilities*. Mobile devices include many mature components, but as complete architectures the current mobile devices are relatively new platforms. These platforms also continue to rapidly evolve. This makes them potentially easier targets, since attackers are likely to find significant unaddressed vulnerabilities in a new platform or new functionality.

## 3. Mobility – Key issues

The following are key issues in mobility. Some of these were noted above as differentiators, but this section discusses them in more detail.

### a. Market Leaders

At the time of writing this report, the key market leaders for mobile platform operating systems are Google (through Android) and Apple (through iOS), with small shares held by Blackberry Limited and Microsoft. The numbers vary month to month and also vary significantly based on the sample set (e.g., by region, by device type, and by type of user).

The following table shows the market share percentages of smartphone operating systems sold worldwide in specific quarters of 2015 as determined by Gartner [Gartner 2015] and IDC [IDC 2015]. Gartner states that global sales of smartphones to end users in 3Q15 totaled 353 million units. These are similar to previous data from 2013 [Gartner2013] [IDC 2013].

**Worldwide Smartphone Sales to End Users by Operating System**

| Operating System | 3Q15 Market Share (Gartner) | 2Q15 Market Share (IDC) |
|---|---|---|
| Android | 84.7% | 82.0% |
| iOS | 13.1% | 13.9% |
| Microsoft | 1.7% | 2.6% |
| BlackBerry | 0.3% | 0.3% |
| Others | 0.3% | 0.4% |

These numbers do not tell the whole story; the enterprise view is different. Apple iOS is more common in enterprises than these numbers suggest, although iOS's dominance has slowly decreased over the years as Android has become more common in enterprises. The Good Technology Mobility Index Report Q2 2015 reported that among the activations for its enterprise product, iOS's overall market share was 64% (compared to 70% before) and Android grew to 32% (compared to 26% before), Windows was 3% and Windows Phone was 1% [Good 2015]. The previous Good Technology Mobility Index Report Q2 2014 reported that the total number of activations was 88% for iOS and 12% for Android; for that one quarter iOS activations were 67% and Android device activations were 32% [Good 2014]. ZDNet examined the 3Q2012 data from Citrix Zenprise as a proxy for enterprise use, and found that in North America the market shares were iOS 55%, Android 41%, and Windows mobile[18] 4%. [Dignan 2013]. Enterprise file sharing and hybrid cloud storage company Egnyte determined that in 1Q2013, iPhone

---

[18] It is not entirely clear, but it appears that this particular source did not distinguish between "Windows mobile" and "Windows phone" and instead merged them together into the category "Windows mobile." This is unfortunate, since they are not the same. Windows Phone is incompatible with Windows Mobile devices and software. Microsoft announced in February 2010 that Windows Phone will supersede Windows Mobile, deprecating Windows Mobile.

had 48%, iPad 30%, and Android (phones and tablets) were 22% [Lomas 2013]. One report notes that on average Android has far more downloads per app than iOS (60,000 vs. 40,000), but that developer revenue per app is more for iOS than Android (the average Android app download brings 2 cents to its developer, while Apple brings in 10 cents) [Louis 2013]. The reason for this discrepancy is that many Android sales are low-end smartphones [Bradley 2013], and the evidence above suggests that these low-end devices are less likely to be used for enterprise work.

Other statistics involving mobility are available via MobiForge [MobiForge].

### b. Apple and Google Approaches

Apple (iOS) and Google (Android) have different business models and approaches. Apple is heavily vertically integrated in hardware, operating system, middleware, and a standard set of applications. Google Android is more like an ecosystem, with multiple suppliers of hardware, a single initial source of operating system and middleware (although suppliers are able customize and contribute to it), and a standard set of applications from Google that is often supplemented by hardware suppliers.

Apple has a more centralized and controlled environment. Apple strictly controls the hardware components, iOS operating system, frameworks, and the APIs onto which applications interface. Apple chooses the suppliers it uses, sets the supplier assurance criteria for their suppliers, and implements this criteria set. For example, applications go through a vetting process before they are accepted into the Apple store for purchase. Apple does not reveal many aspects of its vertically integrated solution, including details of how it evaluates components (including applications) for acceptance, but it is known that Apple does evaluate applications submitted to the Apple store. A developer submits the application, and is provided a report/feedback as to whether or not the application is approved for acceptance into the Apple application store. There is limited visibility into the level of rigor in the Apple evaluation and certification process for application approval. In particular, the detailed testing method, and the rigor to which these tests are administered, is not revealed.

Android on the other hand has an ecosystem of hardware with multiple suppliers providing hardware solutions. It has an open source operating system, frameworks, and APIs that the applications developer community must use as their foundation of applications development and delivery. For supply chain assurances and for the purpose of software assurance, both must be considered. Thus, enterprises chose which hardware to use, which then implies a software stack associated with that hardware, which then in turn is used to install and run applications.

In the case of the Android hardware platform providers (such as Samsung and Motorola), the supplier of the hardware platform (smart phone or tablet) is responsible for

ensuring that the operating system, frameworks, and APIs are further integrated into the hardware solution with appropriate drivers (in most cases provided by the hardware supplier). Hardware platform providers often add additional applications as well. Each platform provider has its own set of designs, suppliers, integration processes, test processes, and acceptance criteria for their platform.

The Android operating system, frameworks, and APIs are based on open source software and allow for community contribution. However, the software stack used in many mobile devices is significantly controlled by Google. Google releases Android in two different parts, with significant licensing differences. The first part is the Android Open Source Platform (AOSP) codebase. This provides the basic components of a smartphone operating system such as the operating system kernel and user interface framework. The AOSP is released as open source software, "though it has been criticized for performing the actual development largely behind closed doors." The second part is Google Mobile Services (GMS), also called Google Services, and this portion is proprietary. GMS can itself be primarily divided into two parts, Google Play Services and the Google Play Store. "Google Play Services provides a wealth of [additional] APIs and system services [such as] for Google Maps, Location, [and] in-app purchasing… while the Google Play Store includes a widely-used collection of apps." [Bright 2014] The Google Play services in particular allow Google to directly update many components, and exert additional control over suppliers. Google separately licenses its proprietary GMS; hardware manufacturers can only use this software and certain trademarks if they meet Google's compatibility standards [Google Commerce 2013]. Applications that use GMS Google Play Services will not work on Android systems that lack GMS. Thus, a significant portion of the Android software is open source software, but many Android devices also include a large amount of proprietary software.

Android application developers typically submit their applications to Google Play. Google Play typically posts these submissions within a few hours, and while there are some tests [Mills 2012], this short time only allows time for some basic scanning and overall suggests relatively little rigor. This suggests that Google Play relies primarily on the mobile operating system mechanisms to isolate applications, and it presumes that developers will do essentially all their own testing. It is possible for organizations to create their own app stores; users can typically choose to use these alternative app stores or install software directly.

Federal government research has primarily focused on Google Android. Examples include the DARPA Transformative Apps (TransApps) program, the NIST work on evaluating applications, TaintDroid, and the Security Enhancements for Android program (included in Android 4.3 and now enforced in Android 4.4). We believe this is primarily because the Android platform is far more open to experimentation, making it far easier to perform research and then contribute or deploy results. For example, DARPA TransApps

program manager Doran Michels said that "in 2010, the iPhone was the darling of consumers, but it was a closed platform that we couldn't adapt for our purposes" [Schechter 2013].

Apple iOS and Google Android have historically had different approaches to granting privileges, even though both fundamentally separate applications from each other, but changes to Android have made them more similar. On Apple iOS, most permissions are requested at run-time by the application; the user can then grant or deny the request at that time. That decision will be remembered for reuse later, and users can also change that decision later.

On Google Android, most permissions are listed in a static "manifest" that comes with the application. Users can see what permissions are requested before installing the application. If the device is running Android version 5.1 or lower, or the application targets SDK version 22 or lower, there is an all-or-nothing choice at install time. If the application lists a dangerous permission in its manifest, the user must grant the permission when they install the application; if they do not grant the permission, the system does not install the application at all. However, if the device is running Android 6.0 or higher, and the application targets SDK version 23 or higher, there is no all-or-nothing choice. In this case, the application must list the permissions in the manifest, and it must request each dangerous permission it needs while the application is running. The user can grant or deny each permission, and the application can continue to run with limited capabilities even if the user denies a permission request [Google2016]. This provides users of newer Android devices with finer-grain security decisions if the application supports them.

However, in many cases users will simply blindly accept requests regardless of when they occur.

Note that both iOS and Android are constantly being updated, and that some of these updates add security features. For example iOS version 7 adds several features of interest to the government, adding (1) the ability to control which applications and accounts can be used to open documents and attachments, (2) the ability to automatically connect with a VPN when managed applications are launched, and (3) automatic enabling of data protection (encryption) of all third-party application data [Breeden 2013]. Android version 4.4 adds security features as well; for example, it strengthens application isolation by using security-enhanced Linux (SELinux) in enforcing mode and applies VPNs per user on multi-user devices [Android.com 2013].

The Department of Defense Commercial Mobile Device Implementation Plan [DoD Implementation] states that "a multi-vendor mobile operating system environment for CMDs shall be supported to enable a device-agnostic procurement approach." Thus, it is

likely that in the DoD both Apple iOS and Google Android mobile devices will be present in the short term, and perhaps others as the market evolves.

### c. App (application) Stores

"The largest [application] stores are believed to be the Apple App Store – Apps for iOS handsets only – and Google Play – apps for Android handsets only."  Canalys reported in May 2013 that both Google Play and the Apple App Store each have over 800,000 applications.  There are other app stores; GetJar claims to be the largest independent app store as of September 2012 (not tied to one operating system, device provider, or carrier), with 600,000 mobile applications.  Some device manufacturers (such as Samsung) and carriers also have app stores. "The Apple App Store enjoys a monopoly over apps for iOS handsets, unless consumers use jailbreak apps to break the restrictions Apple places on how they use their handsets."  Note that enterprises can enter agreements with Apple so their users can get enterprise apps on their iOS devices. "Google Play is not a monopoly, but benefits from Android handsets being the most popular type of smartphone – Google Play is also the default store, coming pre-installed on masses of Android handsets."  [Mobiforge]

Application stores are repositories of application; their owners can determine the criteria for accepting and removing an application.  Third-party application stores can be centralized or distributed, rigorous or not.  In Android, "franchise" applications stores are already happening.

The Department of Defense Commercial Mobile Device Implementation Plan [DoD Implementation] includes plans to establish an enterprise Mobile Application Store (MAS) capability that operates in conjunction with an MDM system, and Mobile applications may be acquired and managed by each DoD Component.  In particular, "completed and approved mobile applications will be able to be downloaded on demand from enterprise and/or DoD Component [mobile application store(s) (MASs)]."

### d. Limited Resources

Mobile devices have limited resources compared to modern desktops and laptops in terms of CPU processor performance, memory, storage, communication channel bandwidth (which may also be unreliable), and battery power.

The limited resources (lower CPU processor performance, memory, storage, and communication channel bandwidth) mean that in many cases, mobile applications must today be specially developed to work on mobile devices.  Many people use web browsers on mobile devices, and there are many mobile web applications.  Mobile web applications, typically implemented using Javascript and HTML5, are typically more portable across different types of mobile devices.  However, there are significant

limitations when using mobile web applications and other portable approaches. For example:

- Memory management is much more difficult on today's mobile devices, and mobile application developers must often "spend a lot of time thinking about memory management." Automated garbage collectors work well if you have at least six times as much memory as needed, but efficiency can greatly harmed if there is less than four times as much memory. "iOS has formed a culture around doing most things manually and trying to make the compiler do some of the easy parts. Android has formed a culture around improving a garbage collector that they try very hard not to use in practice. But either way, everybody spends a lot of time thinking about memory management when they write mobile applications. There's just no substitute for thinking about memory." [Crawford 2013]. Automated garbage collection is deprecated in OS X Mountain Lion v10.8, and will be removed in a future version of OS X; Automatic Reference Counting (ARC) is the recommended approach instead. ARC is supported in Xcode 4.2 for OS X v10.6 and v10.7 (64-bit applications) and for iOS 4 and iOS 5 [Apple ARC].

- Mobile web applications do not have access to all the resources of a mobile device that a platform-specific mobile app would have [Heath 2013].

It is difficult to counter covert channels on a mobile device. Mobile devices have limited resources to devote to covert channel defenses, but adequate resources to implement covert channels at significant bit rates. However, in practice, malicious applications typically do not need to resort to covert channels to perform unauthorized sharing of data. We include countering covert channels as a technical objective that might be selected, but, in practice, many other channels are often available to attackers, and countering those is necessary before covert channels are even relevant.

Mobile devices have limited electrical power, and since they often run on batteries, the device will shut down when the power runs out. This limited power makes it more difficult to simply add memory or CPU horsepower, since these additional facilities may draw additional power. This limited electrical power can even form the basis of an attack; an attacker can in some cases devise an application to drain battery power when the user most needs the device. The power can be consumed by the display (and even color matters), communication channels, and so on. [Murmuria] presents a general methodology for collecting measurements and modelling power usage on smartphones, and presents a power usage model. For example, in their tests they found that blue pixels cause a higher rate of current discharge than green pixels (and green more than red). NIST has performed a number of tests of Android applications, including their power characteristics, though in their tests red consumed the most power [Rausnitz 2013]. Typical 4G implementations can drain power rapidly [Bartlett 2012].

Note that since mobile devices have limited electrical power and must often connect to power to charge, users have a tendency to connect to power sources wherever they are available (e.g., airports, kiosks, etc.). Many of these connectors are USB connectors, which can also perform data transfers. One threat is that attackers may use these USB connectors that users think are only for power as a means to attack the device. Some products are available that only connect the USB power connectors, not the USB data connectors, countering this threat. Another mitigation approach is for users to always charge by connecting directly to wall AC power.

### e. Sensors

Typical mobile devices include a large number of sensors and actuators that can create additional risks. In particular, sensors can provide information about the user, user activities, or user environment to those who should not receive this information. Particularly in the DoD environment, care needs to be taken to ensure that these sensors are controlled and managed. What's more, many sensors are controlled solely by software, or by hardware interlocks that can be worked around, instead of being a simple physical mechanism that disables the sensor in a way that cannot be overridden (for a discussion focusing on laptops, see [Soltani 2013]).

Most obviously, the microphone and camera create an excellent way to capture what someone hears, says, sees, or does, especially when combined with GPS (to identify location) and communication mechanisms (which allow transmission of that information immediately or later). This would enable adversaries to know who is where and what the United States and its allies are trying to do.

There are less obvious uses of these various sensors, however. For example:

- Acoustic cryptoanalysis uses audio information to break cryptographic algorithms. One paper demonstrated using a mobile phone to acoustically extract a full 4096-bit RSA decryption key from a laptop at a distance of 30cm, within an hour [Genkin 2013]. The malicious application could be run on an adversary's phone, or could also be a Trojan horse running on the victim's phone. This could occur anywhere, even in an airport or coffee shop, and is not an attack most people would even think about.

- Similarly, another paper showed that smartphone accelerometers can be used as a high-bandwidth side channel. In particular, the accelerometer sensor could be used by itself to determine the user "tap and gesture-based input as required to unlock smartphones using a PIN/password or Android's graphical password pattern." [Aviv 2012]

In many cases, these misuses of sensors could be implemented as a Trojan horse as part of a larger authorized application. What's more, if the data is not analyzed locally,

and is instead sent elsewhere in raw form, it may be difficult to determine whether the data is being exploited (and if so, how).

# Appendix G.  Additions since the 2013 SOAR

**Additions in 2014**

The version of the paper released in May 2014 extended the version of August 19, 2013.  In particular, it added information specifically focused on mobile platforms (e.g., smartphones and tablets running on operating systems such as iOS and Android). Appendix F discusses the mobile environment, and is wholly new in this version of the paper.

Major changes made in the 2014 revision of the original document included various incremental improvements, including those suggested by reviewer comments from the Sponsors, the Software Engineering Institute (SEI) and the MITRE Corporation.  In particular, the set of technical objectives was expanded and slightly reorganized per reviewer comments.[19]  Other clarifications were made in response to reviewer comments; for example, we now make explicit that unless the paper says otherwise, tools and techniques are automated (and not exclusively manual).

In the technical objectives (of section 4), we:

- Clarified that "permissions, privileges, and access control" included granting resource access to another component that should not be allowed that access. Excessive grants are a problem in any system, but mobile environments often isolate applications from each other by default, and users typically depend on this.  This means that excessive grants can create vulnerabilities unexpected by mobile system users.

- Modified the top-level category, "provide anti-tamper," to also cover "ensure transparency."  The issue of ensuring transparency is of special additional concern in mobile environments; there are often very short time limits for the analysis of mobile software, so tools that deliberately inhibit transparency can make it extremely difficult to ascertain the risk of using third-party applications.

- Added countering excessive power consumption as a key potential technical objective.  Excessive power consumption can cause degradation of server

---

[19] In particular, per MITRE comments we have added countering the use of insufficiently random values (Common Weakness Enumeration (CWE)-330), countering improper certificate validation (CWE-295), and countering excessive iteration (CWE-834).  The matrix itself has also been modified to improve CWE mapping, e.g., to map insufficient compartmentalization to CWE-653.

performance, but in the mobile environment it can lead to complete denial of service by the device.

We did not need to add a new technical objective to cover embedded malicious logic such as Trojan horses (additional functionality not desired by user). However, it is worth noting that in mobile devices this additional functionality can include misuse of sensors and actuators beyond the functionality expected by the user. For example, a mobile application might be granted access to a microphone and the network; that application could then misuse these privileges to record and transmit sound nearby without the user's knowledge or consent.

A number of tool/technology types were added. Major additions (which resulted in changes in section 5 and Appendix C) were:

- *Inter-application flow analyzer*. Since mobile devices isolate applications by default, the flow of data between applications can be even more critical since it is an important mechanism that attackers use to subvert mobile devices.

- *Host application interface scanner*. Application interfaces present an attack surface, yet on mobile devices this attack surface is less obvious than on other types of systems. These tools provide such information.

- *Compare binary/bytecode to application permission manifest*, *permission manifest analyzer*, and *execute and compare with application manifest*. Android applications include a static "manifest" of privileges that the application wishes to use. Several tool/technology types have been created that specifically use and analyze this information. There are some systems outside the mobility environment that also support the static definitions of permissions for an application (e.g., Secure-Enhanced Linux), but the widespread availability of such data with the application itself makes these kinds of approaches especially attractive for tool development.

- *Obfuscated code detection*. Mobile software is often updated rapidly, leading to a need to rapidly evaluate such software as noted above. Obfuscation may be used to counter reverse-engineering of critical or proprietary technology, but it can also be used to counter or slow analysis by other assurance tools. Thus, obfuscated code may create an increased risk of unintentionally vulnerable or intentionally malicious code.

- *Obfuscator*. Mobile devices make it easy to provide software to capture and analyze data, but adversaries would be able to exploit it if they could determine how it works. There are uses for these in non-mobile environments, but in many other situations there are often physical protections as well (e.g., the computer

may be in a locked protected room), while for a mobile device many physical protections are impractical.

- *Framework-based fuzzer*. A framework-based fuzzer can be used in many environments, but the rich fixed framework in a typical mobile environment (e.g., Android and iOS) makes it easier to create a single framework-based fuzzer that can be reapplied to a large number of applications.

- *Automated monitored execution*. Automated monitored execution can be applied in both mobile and non-mobile settings. The limited time often available to analyze third-party mobile applications, and the single fixed framework for a given mobile device, makes automated monitored execution especially attractive for analysis of third-party mobile applications. The paper was later revised to name this automated detonation chamber.

- *Forced path execution*. Most dynamic analysis approaches have the disadvantage that they only execute certain paths (the ones triggered by given inputs). An alternate approach is to test by forcing an application to use other paths, even if the input did not trigger it, to see if following that path is likely to lead to other problems. This approach risks additional false positives, but it can be done relatively quickly. This is especially relevant for evaluating mobile applications, since in many cases organizations wish to evaluate mobile applications quickly even at the cost of the loss of some precision.

- *Man-in-the-middle attack tool*. Mobile devices are almost always connecting wirelessly, making man-in-the-middle attacks easier to perform, so tools focused on detecting such vulnerabilities are even more useful.

- *Track sensitive data*. Mobile devices often include a great deal of sensitive data (including personal data), yet because applications are supposed to be isolated from each other, users tend to assume that sensitive data will not be leaked in an unauthorized way. Yet these data can leak out anyway, through multiple applications; tools designed specifically to look for this thus become more compelling.

The discussion on gaps (in section 9) was extended to include gaps specific to mobility. Other sections of this paper were also modified, e.g., to list the additional people we interviewed and documents we cited.

**Additions in 2016**

This document was further extended in 2016. Key changes from the 2014 version are described here.

In the May 2014 version we noted that there was a lack of specific quantitative data to support the hypothesis that higher software quality tends to produce more secure software. At the time this was a plausible hypothesis that a number of experts believed to be true. Some of the arguments for why this might be true are that:

1. Higher-quality software should have fewer defects, and security vulnerabilities are a subset of software defects. If the percentage of security vulnerabilities is similar in software that is higher quality in general, then software that is higher quality in general would have fewer vulnerabilities. This is not *necessarily* true; it could be that tools and techniques for addressing generic quality defects would leave most security defects unaddressed.

2. Higher-quality software tends to be simpler for tools and humans to analyze, resulting in improved identification of vulnerabilities.

3. Tools designed to look for quality defects may also look for some of the same properties that vulnerability-finding tools look for, and thus they really are not distinct.

However, many seemingly reasonable hypotheses are false. We believed in 2014 that it was important to investigate this claim before recommending it. This question is important, because if it is true, then it might be appropriate to first use tools to identify quality problems, fix the problems they identify, and then use other tools for more complex analysis. More evidence that supports this hypothesis has since been published. In particular, SEI [Woody 2014] published in December 2014 a compendium of evidence to support the claim that higher quality software tends to produce more secure software.

While more evidence would be welcome, we believe the preponderance of evidence now is that improving the general quality of software tends to improve the security of the software. This does **not** mean that using only generic quality tools is enough to develop secure software. Instead, it means that using generic quality tools can be a valuable aid in developing secure software.

We searched for new types of tools and techniques. Software assurance is not a solved problem, and while most tool suppliers had refined their tools further, we were disappointed that we did not find more new approaches. That said, we added three new tool categories not in previous versions of the SOAR: "coverage-guided fuzz tester," "probe-based attacks with tracked flow," and "track data and control flow." These are defined as:

- A coverage-guided fuzz tester is a fuzz tester that uses code coverage information to determine new inputs to test.

- The track data and control flow tool/technique tracks data and control flows from inputs and other data sources to data sinks, and reports when rules (predefined or user defined) are triggered indicating a potential vulnerability.

- The probe-based attack with tracked flow tool/technique observes normal behavior while tracking data and control flows within the program (possibly through several tiers), sends probing inputs to determine patterns of behavior that might indicate a potential vulnerability, and then based on these patterns, performs simulated attacks to identify actual vulnerabilities.

All of these additional types of tools are hybrid approaches, which is interesting because we had previously predicted that more tool types would be created as hybrids to take advantage of the information available from both static and dynamic analysis. All of these tool types have great promise for detecting vulnerabilities in applications. As for the other types, we have estimated their effectiveness for various technical objectives in Appendix E. Note that since we have less information on these newer types of tools, their values in Appendix E are more subject to future change.

We added a new major heading, "Application," to provide specific guidance on how to apply this information and a wholly new section with tips on selecting technical objectives for a system, as well as an expansion on how to select tools and techniques given those technical objectives. We also updated the vignettes to match.

Other key changes include:

1. We renamed "automated monitored execution (limited time)" to "automated detonation chamber (limited time)" because this is a more precise description.

2. We renamed the technical objective "counter known vulnerabilities" to "counter known unintentional-like vulnerabilities"; this means that the known intentional-like vulnerabilities are uniquely separated into the technical objective "counter intentional-"like"/malicious logic." This changed the entry for the tool type "traditional virus/ spyware scanner" which is now more clearly allocated to the latter technical objective.

3. In the large table of Appendix E, we added a column to map to the OWASP top 10 of 2013.

4. We briefly discussed other kinds of tools that are related but not the primary focus of this paper. These include SwA correlation tools, as well as various excluded tools and techniques (general-purpose software test tools and test frameworks, combinatorial testing, and threat intelligence).

5. We compare our high-level tool grouping (static, dynamic, and hybrid) to Gartner's, since some people may be familiar with Gartner's [Mello2015].

Gartner uses the general term *application security testing* (AST), as well as the term *static AST* (SAST), which is basically equivalent to our static group, dynamic AST (DAST), which is basically equivalent to our dynamic group, interactive AST (IAST), which we term as hybrid. They separately list mobile AST, which we do not.

6. We updated the tables and other information related to mobile devices. In particular, the latest version of Android has changes to its application security model, which we discuss.

# Acronyms

| | |
|---|---|
| AC | alternating current |
| ACM | Association for Computing Machinery |
| ACSAC | Annual Computer Security Applications Conference |
| ADT | Action for Technological Development |
| AJAX | Asynchronous JavaScript and XML |
| AOSP | Android Open Source Platform |
| ARC | Automatic Reference Counting |
| ASACoE | Application Software Assurance Center of Excellence |
| ASCAD | Adelard Safety Case Development |
| ASCE | (Adelard) Assurance and Safety Case Environment[20] |
| ASIC | Application Specific Integrated Circuits |
| ASLR | Address Space Layout Randomization |
| ASP | Active Server Pages |
| AT&L | Acquisition, Technology, and Logistics |
| | |
| BAH | Booz Allen Hamilton |
| BAT | Binary Analysis Tool |
| BSIMM | Building Security In Maturity Model |
| BYOD | Bring Your Own Device |
| | |
| C&A | Certification and Accreditation |
| CA | Certificate Authority |
| CAPEC | Common Attack Pattern Enumeration and Classification |
| CAS | Center for Assured Software (part of NSA) |
| CCEVS | Common Criteria Evaluation and Validation Scheme |
| CERT | Not an acronym, but formerly Computer Emergency Readiness Team |
| CFTT | Computer Forensics Tools Testing |
| CIL | Common Intermediate Language |
| CIO | Chief Information Office(r) |
| CLI | Common Language Infrastructure |
| CLM | Component Lifecycle Management |
| CM | Configuration Management |
| CMU | Carnegie Mellon University |
| COBOL | COmmon Business-Oriented Language |
| COFF | Common Object File Format |
| COP | Community of Practice |
| COTS | Commercial Off-The-Shelf |

---

[20] This acronym is defined at http://www.adelard.com/asce/general/graphArgumentation.html.

| | |
|---|---|
| CPI | Critical Program Information |
| CPU | Central Processing Unit |
| CSIAC | Cyber Security and Information Systems Information Analysis Center |
| CTO | Chief Technical Officer |
| CVE | Common Vulnerability Enumeration |
| CVS | Concurrent Versions System |
| CWE | Common Weakness Enumeration |
| | |
| DACS | Data and Analysis Center for Software, consolidated into CSIAC |
| DAG | Defense Acquisition Guidebook |
| DARPA | Defense Advanced Research Projects Agency |
| DASD(SE) | Deputy Assistant Secretary of Defense (Systems Engineering) |
| DDC | Diverse Double-Compiling |
| DFARS | DoD FAR Supplement |
| DHS | Department of Homeland Security |
| DISA | (DoD) Defense Information Systems Agency |
| DNS | Domain Name System |
| DoD | Department of Defense |
| DoDI | Department of Defense Instruction |
| DoS | Denial of Service |
| | |
| ELF | Executable and Linkable Format (formerly Extensible Linking Format) |
| ENISA | European Network and Information Security Agency |
| EWA | Electronic Warfare Associates, Inc. |
| | |
| FAR | Federal Acquisition Regulation |
| FPGA | Field Programmable Gate Array |
| FTP | File Transmission Protocol |
| FW | FireWall |
| | |
| GAO | Government Accountability Office |
| GMS | Google Mobile Services |
| GNU | GNU's Not Unix |
| GOTS | Government Off-The-Shelf |
| GPU | Graphics Processing Unit |
| GRC | Gibson Research Corporation |
| GSN | Goal Structuring Notation |
| | |
| HCSS | High Confidence Software & Systems |
| HA | High Assurance |
| HKSAR | Hong Kong Special Administrative Region |
| HP | Hewlett-Packard |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol |

| | |
|---|---|
| IA | Information Assurance |
| IATAC | Information Assurance TAC, consolidated into CSIAC |
| IAVA | Information Assurance Vulnerability Alert |
| IBM | International Business Machines |
| IDA | Institute for Defense Analyses *or* part of name of IDA Pro |
| IDE | Integrated Development Environment |
| IDS | Intrusion Detection System |
| IEEE | Institute of Electrical and Electronics Engineers |
| IIT | Information and Infrastructure Technologies (part of EWA) |
| IP | Intellectual Property *or* Internet Protocol |
| IPS | Intrusion Prevention System |
| | |
| JAR | Java ARchive (format) |
| JVM | Java Virtual Machine |
| | |
| K | one-thousand |
| | |
| LTE | Long-Term Evolution |
| | |
| MAM | Mobile Application Management |
| MAS | Mobile Application Store |
| MDM | Mobile Device Management |
| MSDN | Microsoft Developer Network (MSDN) |
| | |
| NDAA | National Defense Authorization Act |
| NDI | Non-developmental item |
| NDIA | National Defense Industrial Association |
| NIAP | National Information Assurance Partnership |
| NIST | National Institute of Standards and Technology |
| NSA | National Security Agency |
| NUL | Null character |
| NVD | National Vulnerability Database |
| | |
| O-TTPF | Open Trusted Technology Provider Framework |
| OASD(HA) | Office of the Assistant Secretary of Defense for Health Affairs |
| OCIL | Open Checklist Interactive Language |
| OISF | Open Information Security Foundation |
| OS | Operating System |
| OSD | Office of the Secretary of Defense |
| OSS | Open Source Software (note that nearly all OSS is COTS) |
| OTS | Off-the-shelf |
| OUSD(AT&L) | Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics |
| OVAL | Open Vulnerability and Assessment Language |
| OWASP | Open Web Application Security Project |

| | |
|---|---|
| PDF | Portable Document Format |
| PE | (Microsoft) Portable Executable |
| PHP | PHP: Hypertext Preprocessor (formerly Personal Home Page) |
| PI | Principle Investigator |
| PL/1 | Programming Language/1 |
| PL/SQL | Procedural Language/Structured Query Language |
| PM | Program Manager |
| PMD | (Not an acronym) |
| PMO | Program Management Office |
| POSIX | Portable Operating System Interface |
| PPP | Program Protection Plan |
| | |
| RSA | (Ron) Rivest – (Adi) Shamir – (Leonard) Adleman |
| RTOS | Real-Time Operating System |
| | |
| SaaS | Software-as-a-Service |
| SAMATE | Software Assurance Metrics And Tool Evaluation |
| SANS | System Administration, Networking, and Security |
| SAST | Static Application Security Testing |
| SATE | Static Analysis Tool Exposition |
| SBIR | Small Business Innovation Research |
| SCAP | Security Content Automation Protocol |
| SCRM | Supply Chain Risk Management |
| SCWA | Source Code Weakness Analysis |
| SDLC | Software Development Lifecycle |
| SE | Systems Engineering |
| SEI | Software Engineering Institute |
| SIEM | Security Information and Event Management |
| SME | Subject Matter Expert |
| SOAR | State-of-the-Art Resource(s) |
| SQL | Structured Query Language |
| SRG | Security Requirements Guide |
| SSH | Secure Shell, sometimes known as Secure Socket Shell |
| SSL | Secure Sockets Layer |
| STIG | Security Technical Implementation Guide |
| STONESOUP | Securely Taking On New Executable Software of Uncertain Provenance |
| STRIDE | Spoofing, Tampering, Repudiation, Information disclosure, DoS, Elevation of privilege |
| SwA | Software Assurance |
| | |
| TAC | Technology Analysis Center |
| TCB | Trusted Computing Base |
| TCP | Transmission Control Protocol |
| TCP/IP | Internet protocol suite, including TCP, UDP, IP, and DNS |
| TechSgt | Technical Sergeant |
| TFS | (Microsoft) Team Foundation Server |

| | |
|---|---|
| TLS | Transport Layer Security |
| TMA | TRICARE Management Activity |
| TOE | Target of Evaluation (the software being evaluated) |
| TSN | Trusted Systems and Networks |
| | |
| UDP | User Datagram Protocol |
| UK | United Kingdom |
| URL | Uniform Resource Locator |
| U.S. | United States |
| USB | Universal Serial Bus |
| | |
| VB6 | Visual Basic 6 |
| VBNET | Visual Basic for .NET |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very-High-Speed Integrated Circuit |
| VSS | Visual SourceSafe |
| | |
| WASC | Web Application Security Consortium |
| | |
| XCCDF | eXensible Configuration Checklist Description Format |
| XML | eXtensible Markup Language |

All trademarks are owned by their respective trademark holders.

# Bibliography

[Agematsu 2012] Agematsu, Harunobu, Junya Kani, Kohei Nasaka, Masakatsu Nishigaki; Hideaki Kawabata, Takamasa Isohara, and Keisuke Takemori. "A Proposal to Realize the Provision of Secure Android Applications." *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*. July 4-6, 2012. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6296936.

[Alazan 2012] Alazab, Moutaz, Veelsha Moonsamy, Lynn Batten, Ronghua Tian, and Patrik Lantz. "Analysis of Malicious and Benign Android Applications." *2012 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. June 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6258212.

[Amadeo 2013] Amadeo, Ron. "Google's iron grip on Android: Controlling open source by any means necessary." *Ars Technica*. October 2013. http://arstechnica.com/gadgets/2013/10/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/.

[Android 2013] Android.com. "Security Enhancements in Android 4.4." 2013. http://source.android.com/devices/tech/security/enhancements44.html.

[Android Fundamentals 2013] *Application Fundamentals*. 2013. https://developer.android.com/guide/components/fundamentals.html.

[Android intents] "Intents and Intent Filters". *API Guide*. http://developer.android.com/guide/components/intents-filters.html.

[Apple ARC 2013] Apple Automatic Reference Counting (ARC). "Transitioning to ARC Release Notes." August 2013, https://developer.apple.com/library/mac/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html.

[Apple 2012] Apple Inc, *iOS Security*. May 2012. http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf.

[AT&T & McAfee 2011] AT&T and McAfee. *Keep Your Enemies Closer: Three Steps to Bring Mobile Devices into Your Security Infrastructure*. 2011. http://www.mcafee.com/us/resources/white-papers/wp-keep-your-enemies-closer.pdf.

[Aviv 2012] Aviv, Adam J., Benjamin Sapp, Matt Blaze and Jonathan M. Smith. "Practicality of Accelerometer Side Channels on Smartphones." *Annual Computer Security Applications Conference (ACSAC)*. February 2012.

http://www.acsac.org/2012/openconf/modules/request.php?module=oc_program&action=view.php&a=&id=56&type=4&OPENCONF=75799905b2bb79c800ddf7ade3caad00.

[BAH 2009] Booz Allen Hamilton. *Software Security Assessment Tools Review*. March 2, 2009. http://samate.nist.gov/docs/NAVSEA-Tools-Paper-2009-03-02.pdf.

[Barmier] Barmier, Tal. *Mobile Test Automation Trends & Tools Evaluation Criteria*. http://d242m5chux1g9j.cloudfront.net/Mobile%20Test%20Automation_Overview%20&%20Tools%20Evaluation.pdf.

[Bartlett 2012] Bartlett, Mitch. "Why Does 4G Really Drain Your Battery?" *PhoneTipz*. January 2012. http://phonetipz.com/why-does-4g-really-drain-your-battery.

[Black 2008] Black, Paul E., Elizabeth Fong, Vadim Okun, and Romain Gaucher. *Software Assurance Tools: Web Application Security Scanner Functional Specification*. Version 1.0. January 2008. http://samate.nist.gov/docs/webapp_scanner_spec_sp500-269.pdf.

[Böck] Böck, Hanno. "Part 2: Find more Bugs with Address Sanitizer". *The Fuzzing Project (Tutorial)*. https://fuzzing-project.org/tutorial2.html. Retrieved March 7, 2016.

[Bradley 2013] Bradley, Tony. "Android Dominates Market Share, But Apple Makes All The Money." Forbes. November 2013. http://www.forbes.com/sites/tonybradley/2013/11/15/android-dominates-market-share-but-apple-makes-all-the-money/.

[Breeden 2013] Breeden II, John. "3 reasons iOS 7 could be suitable for government." *GCN* (sic). November 2013. http://digital.gcn.com/?iid=83251#folio=10.

[Bright 2014] Bright, Peter. "Neither Microsoft, Nokia, nor anyone else should fork Android. It's unforkable." Ars Technica. February 8, 2014. http://arstechnica.com/information-technology/2014/02/neither-microsoft-nokia-nor-anyone-else-should-fork-android-its-unforkable/

[Bulgeisi 2013] Bugliesi, Michele, Stefano Calzavara, and Alvise Spanò. "Lintent: towards security type-checking of Android applications." *Formal Techniques for Distributed Systems, Lecture Notes in Computer Science Volume 7892*. pp 289-304. 2013. http://www.dais.unive.it/~calzavara/papers/forte13.pdf.

[Butler] Butler, Ricky. "What is Formal Methods?" (sic). *NASA Langley Formal Methods Site*. http://shemesh.larc.nasa.gov/fm/fm-what.html.

[CAS 2011] National Security Agency (NSA) Center for Assured Software (CAS). *CAS Static Analysis Tool Study – Methodology*. 2011. http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf.

[CAS 2012] National Security Agency (NSA) Center for Assured Software (CAS), *CAS Static Analysis Tool Study – Methodology*. 2012.

http://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.

[CAS 10x10 2013] National Security Agency (NSA) Center for Assured Software (CAS). *10x10 Project Report*. April 2013.

[CAS Survey 2013] National Security Agency (NSA) Center for Assured Software (CAS). *Mobile Software Assurance Analysis Tool Survey*. April 2013.

[CAS Vuln 2013] National Security Agency (NSA) Center for Assured Software (CAS). *Android A to C Application Vulnerabilities*. April 2013.

[Cardinal 2013] David Cardinal. "Is Play Google's new secret weapon against Android fragmentation?" May 16, 2013. *Extreme Tech*. http://www.extremetech.com/gaming/156048-is-play-googles-new-secret-weapon-against-android-fragmentation.

[Carrier 2003] Carrier, Brian. "Defining Digital Forensic Examination and Analysis Tools Using Abstraction Layers." International Journal of Digital Evidence. Winter 2003. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9813&rep=rep1&type=pdf.

[Chen 2012] Chen, Shay. "The 2012 Web Application Scanner Benchmark." *Security Tools Benchmarking*. July 2012. http://sectooladdict.blogspot.com/2012/07/2012-web-application-scanner-benchmark.html.

[Chen 2014] Chen, Shay. "WAVSEP Web Application Scanner Benchmark 2014." *Security Tools Benchmarking*. February 5, 2014. http://sectooladdict.blogspot.com/2014/02/wavsep-web-application-scanner.html

[Chin 2011] Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing Inter-Application Communication in Android." *MobiSys '11*. June 28–July 1, 2011. http://www.cs.berkeley.edu/~afelt/intentsecurity-mobisys.pdf.

[Chmielewski 2013] Chmielewski, Clift, Fonrobert, and Ostwald. "Find and Fix Vulnerabilities Before Your Application Ships." *MSDN Magazine*. Microsoft. 2013. http://msdn.microsoft.com/en-us/magazine/cc163312.aspx.

[Cigital] Cigital. Securing Mobile Applications. http://www.cigital.com/resources/downloads/?dl=SecuringMobileApps.

[CIO Council 2013] Federal CIO Council. "The Potential of Mobile Apps." May 2013. https://cio.gov/the-potential-of-mobile-apps/.

[CITO 2012] CITO Research. "The Lessons of Google Play for API Designers and Enterprise Architects." 2012. http://www.citoresearch.com/app-dev/lessons-google-play-api-designers-and-enterprise-architects#sthash.IbQwiJ99.dpuf.

[Congress 2013] U.S. Congress, Jan. 2, 2013, National Defense Authorization Act (NDAA) for fiscal year 2013. http://www.gpo.gov/fdsys/pkg/PLAW-112publ239/pdf/PLAW-112publ239.pdf

[CNSS2015] Committee on National Security Systems (CNSS). April 6, 2015. *National Information Assurance Glossary*. CNSS Instruction No. 4009.

[Crawford 2013] Crawford, Drew. "Why mobile web apps are slow." July 2013. http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/.

[Chun 2012] Chun, Woo-Sung and Dea-Woo Park. "Malicious Code Hiding Android Apps Distribution and Hacking Attacks and Incident Analysis." 2012 8th International Conference on Information Science and Digital Content Technology (ICIDT) Volume 3. June 26-28, 2012. Pp. 686–689. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=626936[DAG] Department of Defense (DoD). *Defense Acquisition Guidebook (DAG)*. https://dag.dau.mil/Pages/Default.aspx.

[DASD(SE) 2011] DASD(SE). *Program Protection Plan Outline and Guidance. Version 1.0.* July 2011. http://www.acq.osd.mil/se/docs/PPP-Outline-and-Guidance-v1-July2011.pdf

[DASD(SE) 2014] DASD(SE). *Software Assurance Countermeasures in Program Protection Planning Protection.* March 2014. http://www.acq.osd.mil/se/docs/SwA-CM-in-PPP.pdf

[Dai Zovi] Dai Zovi, Dino A. "Apple iOS 4 Security Evaluation." http://www.trailofbits.com/resources/ios4_security_evaluation_paper.pdf

[DARPA 2013] Michel, Doran. "Transformative Apps." 2013. http://www.darpa.mil/Our_Work/I2O/Programs/Transformative_Apps.aspx.

[Datta 2012] Datta, Soumya Kanti, Christian Bonnet, Navid Nikaein. "Android Power Management: Current and Future Trends." June 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6311253&queryText%3DAndroid+Power+Management%3A+Current+and+Future+Trends.

[DHS Acq 2012] DHS, Software Assurance in Acquisition and Contracting Language. May 18, 2012. https://buildsecurityin.us-cert.gov/sites/default/files/publications/AcquisitionAndContractLanguage_PocketGuideV1%202_05182012_PostOnline.pdf.

[Dignan 2013] Dignan, Larry. "Android, Apple iOS flip consumer, corporate market share." February 2013. http://www.zdnet.com/android-apple-ios-flip-consumer-corporate-market-share-7000011255.

[DHS 2013] Roll Call Release, "Threats to Mobile Devices Using the Android Operating System." July 2013. http://www.fiercemobilegovernment.com/story/dhs-fbi-warn-android-malware-threat/2013-08-28#ixzz2dINgDwaC.

[DISA STIG 2013] DISA STIG MDM. "Mobile Device Management (MDM) Security Requirements Guide (SRG)." January 2013. http://iase.disa.mil/stigs/a-z.html.

[DISA] DoD Mobility PMO. "DISA Mobility program." http://www.disa.mil/Services/Enterprise-Services/Mobility.

[Dr. Dobbs 2007] "SIEM: A Market Snapshot." February 2007. http://www.drdobbs.com/siem-a-market-snapshot/197002909.

[DoD 2009] DoD. October 16, 2009. Clarifying Guidance Regarding Open Source Software (OSS). http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf

[DoD CIO] Department of Defense (DoD) Chief Information Officer (CIO). "Department of Defense Commercial Mobile Device Implementation Plan." February 2013. http://www.defense.gov/news/dodcMdimplementationplan.pdf.

[DoD CIO 2012] Department of Defense (DoD) Chief Information Officer (CIO). "Department of Defense Mobile Device Strategy Version 2.0." May 2012. http://www.defense.gov/news/dodmobilitystrategy.pdf.

[DoDI 5000.02] DoD. *Operation of the Defense Acquisition System*. DoD Instruction 5000.02. January 7, 2015. http://www.dtic.mil/whs/directives/corres/pdf/500002p.pdf.

[DoDI 5200.44] DoD. November 5, 2012. *Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN)*. DoD Instruction 5200.44. http://www.dtic.mil/whs/directives/corres/pdf/520044p.pdf.

[DoDI 8500.01] DoD. March 14, 2014. *Cybersecurity*. DoD Instruction 8500.01. http://www.dtic.mil/whs/directives/corres/pdf/850001_2014.pdf

[Dynodroid 2013] Dynodroid project. 2013. "Dynodroid: Automated Testing of Smartphone Apps." http://pag.gatech.edu/dynodroid and http://www.cc.gatech.edu/~naik/dynodroid.html.

[eEye] eEye Digital Security. "Simplifying the Challenges of Mobile Device Security". http://resources.idgenterprise.com/original/AST-0059330_eEye_Mobile_Security_White_Paper.pdf.

[eEye Digital Security] eEye Digital Security. "Best Practices for Securing Remote and Mobile Devices." http://www.eeye.com/eEyeDigitalSecurity/media/White-Papers/Best-Practices-for-Securing-Remote-and-Mobile-Devices-WP.pdf?ext=.pdf.

[Egele] Egele, Manuel, Christopher Kruegel, Engin Kirda, Giovanni Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications." http://www.seclab.tuwien.ac.at/papers/egele-ndss11.pdf.

[Egners 2012] Egners, Andre, Ulrike Meyer, Bjorn Marschollek. "Messing with Android's Permission Model." 2012. http://www.computer.org/csdl/proceedings/trustcom/2012/4745/00/4745a505-abs.html.

[ENISA] European Union Agency for Network and Information Security (ENISA). *Smartphone security*. http://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1.

[Emanuelsson 2008] Emanuelsson, PÄar, and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools (Extended Version)." January 2008. http://liu.diva-portal.org/smash/get/diva2:330560/FULLTEXT01.

[Enck 2010] Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." 2010. http://appanalysis.org/tdroid10.pdf.

[Enck 2011] Enck, William, "Defending Users Against Smartphone Apps: Techniques and Future Directions." *Proceedings of the 7th international conference on Information Systems Security (ICISS 2011)*. 2011. http://www.enck.org/pubs/enck-iciss11.pdf and http://dl.acm.org/citation.cfm?id=2178076.

[Enck 2012] Enck, William. "Analysis Techniques for Mobile Operating System Security." April 2012. https://etda.libraries.psu.edu/paper/11817/6600.

[Ernst & Young, 2012] Ernst and Young. "Mobile Device Security: Understanding Vulnerabilities and Managing Risks." January 2012. http://www.ey.com/Publication/vwLUAssets/Mobile_Device_Security/$FILE/Mobile-security-devices_AU1070.pdf.

[Erturk 2012] Erturk, Emre. "A Case Study in Open Source Software Security and Privacy: Android Adware." June 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6280226&queryText%3DA+Case+Study+in+Open+Source+Software+Security+and+Privacy%3A+Android+Adware.

[Fed CIO Council 2013] Federal CIO Council. "Adoption of Commercial Mobile Applications within the Federal Government Digital Government Strategy Milestone 5.4." May 2013. https://cio.gov/wp-content/uploads/downloads/2013/05/Commercial-Mobile-Application-Adoption-DGS-Milestone-5.4.pdf.

[Fed CIO Mobile Security 2013] Federal CIO Council. "Government Mobile and Wireless Security Baseline. May 2013. https://cio.gov/wp-content/uploads/downloads/2013/05/Federal-Mobile-Security-Baseline.pdf.

[Fed CIO Mobile 2013] Federal CIO Council. "Mobile Computing Decision Framework." May 2013. https://cio.gov/wp-content/uploads/downloads/2013/05/Mobile-Security-Decision-Framework.pdf.

[Fed CIO Arch 2013] Federal CIO Council. "Mobile Security Reference Architecture." May 2013. https://cio.gov/wp-content/uploads/downloads/2013/05/Mobile-Security-Reference-Architecture.pdf.

[Fed CIO Use Case 2013] Federal CIO Council. "Federal Mobile Computing Security Baselines: Moderate Federal Employee Use Case." May 2013. https://cio.gov/wp-content/uploads/downloads/2013/05/Federal-Mobile-Security-Baseline-Appendix-A.xlsx.

[Felt 2011] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. October 2011. "Android Permissions Demystified," http://www.cs.berkeley.edu/~afelt/Android_permissions.pdf.

[GAO-12-757] GAO. "Better Implementation of Controls for Mobile Devices Should Be Encouraged." September 2012. http://www.gao.gov/assets/650/648519.pdf.

[Gartner 2013] Gartner. "Gartner Says Smartphone Sales Accounted for 55 Percent of Overall Mobile Phone Sales in Third Quarter of 2013." November 2013. https://www.gartner.com/newsroom/id/2623415.

[Gartner2015] Gartner. "Gartner Says Emerging Markets Drove Worldwide Smartphone Sales to 15.5 Percent Growth in Third Quarter of 2015." November 18, 2015. http://www.gartner.com/newsroom/id/3169417

[Genkin 2013] Genkin, Daniel, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. 2013. http://www.cs.tau.ac.il/~tromer/acoustic/.

[GMU 2012] Schulte, Brian, Haris Andrianakis, Kun Sun, and Angelos Stavrou. "NetGator: Malware Detection Using Program Interactive Challenges." http://cs.gmu.edu/~astavrou/research/Netgator_DIMVA_2012.pdf.

[Goertzel 2007] Goertzel, Karen Mercedes, et al. Software Security Assurance: A State-of-the-Art Report (SOAR). June 2007. http://iac.dtic.mil/csiac/download/security.pdf.

[Gold 2012] Gold, Steve. "Android: A Secure Future at Last?" March 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6179102&queryText%3DAndroid%3A+A+Secure+Future+at+Last%3F.

[Good 2014] Good Technologies. 2014. (Good) *Mobility Index Report Q2 2014*. https://media.good.com/documents/rpt-mobility-index-q2-2014.pdf

[Good 2015] Good Technologies. 2015. (Good) *Mobility Index Report Q2 2015*. http://venturebeat.com/2015/08/11/ios-slips-to-62-enterprise-share-in-q2-2015-android-hits-32-and-windows-stays-flat-at-4/

[Google Commerce 2013] Google Commerce - Android Developer Blog, "With Google Play services 4.0, US Android developers can now try out Google Wallet Instant Buy API." October 2013. http://googlecommerce.blogspot.com/2013/10/with-google-play-services-40-us-android.html.

[Google2016] "Requesting Permissions at Run Time". Developer.android.com. Retrieved 2016-04-25. http://developer.android.com/training/permissions/requesting.html

[Govt Biz Council] Government Business Council. "Mobile Security: Efforts to Secure Federal Devices." http://cdn.govexec.com/interstitial.html?rf=http%3A%2F%2Fwww.govexec.com%2Fgbc%2Fmobile-security-efforts-secure-federal-devices%2F41004%2F.

[Greenberg, 2013] Greenberg, Adam. "DHS, FBI warn of Android malware threat." August 2013. http://www.scmagazine.com/dhs-fbi-warn-first-responders-of-android-threat/article/308961/.

[GSA] "Helping agencies plan, develop, test and launch anytime, anywhere, any device mobile products and services for the public." http://gsablogs.gsa.gov/dsic/get-it-done/mobile-application-development-program/.

[Guirguis 2003] Guirguis, Ragi. "Network- and Host-Based Vulnerability Assessments: An Introduction to a Cost Effective and Easy to Use Strategy." SANS. June 14, 2003. http://www.sans.org/reading_room/whitepapers/auditing/network-host-based-vulnerability-assessments-introduction-cost-effective-easy_1200.

[Heath 2013] Heath, Nick. "Web apps: the future of the internet, or an impossible dream?" August 2013. http://www.zdnet.com/web-apps-the-future-of-the-internet-or-an-impossible-dream-7000019320/.

[Hernan 2006] Hernan, Shawn, Scott Lambert, Tomasz Ostwald, and Adam Shostack. "Uncover Security Design Flaws Using The STRIDE Approach." *MSDN Magazine*. November 2006. http://msdn.microsoft.com/en-us/magazine/cc163519.aspx,

[HKSAR 2008] Government of the Hong Kong Special Administrative Region (HKSAR). An Overview of Vulnerability Scanners. February 2008. http://www.infosec.gov.hk/english/technical/files/vulnerability.pdf.

[Holland 2004] Holland. "Understanding IPS and IDS: Using IPS and IDS together for Defense in Depth." February 2004. http://www.sans.org/reading_room/whitepapers/detection/understanding-ips-ids-ips-ids-defense-in-depth_1381.

[Hu] Hu, Cuixiong, and Iulian Neamtiu, "Automating GUI Testing for Android Applications." http://www.cs.ucr.edu/~neamtiu/pubs/ast11hu.pdf.

[Huang 2012] Huang, Linya, and Qiaoyan Wen. "The Design and Implementation of Android File Access Control System." October 2012. http://ieeexplore.ieee.org/search/searchresult.jsp?newsearch=true&queryText=The+Design+and+Implementation+of+Android+File+Access+Control+System&x=48&y=12.

[Holwerda 2013] Holwerda, Thom. "The second operating system hiding in every mobile phone." November 2013. http://www.osnews.com/story/27416/The_second_operating_system_hiding_in_every_mobile_phone.

[IATAC 2007] IATAC/DACS. *Software security assurance*. July 31, 2007. http://iac.dtic.mil/csiac/download/security.pdf.

[IEEE 2008] IEEE. *IEEE Standard for Software Reviews and Audits*. 2008. IEEE Std 1028-2008.

[IBM] IBM Software. "Mobility is Moving Fast. To Stay in Control, You Have to Prepare for Change." http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=TIW14132USEN.

[IDC 2013] IDC. "Android Pushes Past 80% Market Share While Windows Phone Shipments Leap 156.0% Year Over Year in the Third Quarter, According to IDC." November 2013. http://www.idc.com/getdoc.jsp?containerId=prUS24442013.

[IDC2015] IDC. "Smartphone OS Market Share, 2015 Q2". http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Retrieved 2016-03-11.

[Johnson] Johnson, Ryan, Zhaohui Wang, Corey Gagnon, Angelos Stavrou. "Analysis of Android Applications' Permissions." http://cs.gmu.edu/~astavrou/research/Analysis%20of%20Android%20Applications%E2%80%99%20Permissions.pdf.

[Kesäniemi 2009] Kesäniemi, Ari, and Nixu Oy. "Automatic vs. Manual Code Analysis." OWASP. November 2009. http://www.owasp.org, https://www.owasp.org/images/5/53/Ari_kesaniemi_nixu_manual-vs-automatic-analysis.pdf.

[Khan 2012] Khan, Sohail, Mohammad Nauman, Abu Talib Othman, and Shahrulniza Musa. "How Secure is your Smartphone: An Analysis of Smartphone Security Mechanisms." 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6246082&queryText%3DHow+Secure+is+your+Smartphone%3A+An+Analysis+of+Smartphone+Security+Mechanisms.

[Knight 1986] Knight, J. C. and Leveson, N. G. "An experimental evaluation of the assumption of independence in multiversion programming." *IEEE Transactions on Software Engineering*. January 1986.

[Koetsier] Koetsier, John. "700K of the 1.2M apps available for iPhone, Android, and Windows are zombies." August 2013. http://venturebeat.com/2013/08/26/700k-of-the-1-2m-apps-available-for-iphone-android-and-windows-are-zombies/.

[Krebs] Krebs, Brian. "Beware of Juice-Jacking." http://krebsonsecurity.com/2011/08/beware-of-juice-jacking/.

[Kupsch] Kupsch, James A., and Barton P. Miller. "Manual vs. Automated Vulnerability Assessment: A Case Study." http://pages.cs.wisc.edu/~kupsch/va/ManVsAutoVulnAssessment.pdf.

[Lanier] Lanier, Zach, and Andrew Reiter (Veracode). "Mapping & Evolution of Android Permissions." http://www.slideshare.net/quineslideshare/mapping-and-evolution-of-Android-permissions.

[Lee2014] Lee, Robert M., "Cyber Threat Intelligence", *The State of Security*, Oct 2, 2014, http://www.tripwire.com/state-of-security/security-data-protection/cyber-threat-intelligence/

[Liu 2011] Liu, Jianye, and Jiankun Yu. "Research on Development of Android Applications." 2011.

[Lomas 2013] Lomas, Natasha. "More Data Showing iOS, Especially The iPhone, Still Killing It In The Enterprise, At Android's Expense." March 2013. http://techcrunch.com/2013/03/07/more-data-showing-ios-and-especially-the-iphone-still-killing-it-in-the-enterprise-at-androids-expense/.

[Lookout] Lookout Mobility. "Lookout Mobility Technical Teardown: DroidDream." 2011. https://blog.lookout.com/wp-content/uploads/2011/03/COMPLETE-DroidDream-Technical-Tear-Down_Lookout-Mobile-Security.pdf.

[MacDonald 2009] MacDonald, Neil (Gartner). "Best Practices: Secure Mobile Development for iOS and Android." July 2009. https://viaforensics.com/resources/reports/best-practices-ios-Android-secure-mobile-development/.

[MacDonald 2009] MacDonald, Neil (Gartner). "Byte Code Analysis is not the same as Binary Analysis." 2009. http://blogs.gartner.com/neil_macdonald/2009/07/24/byte-code-analysis-is-not-the-same-as-binary-analysis/.

[Madden 2012] Madden, Brian. "What is MDM, MAM, and MIM, and what is the difference?" 2012.

http://www.brianmadden.com/blogs/brianmadden/archive/2012/05/29/what-is-mdm-mam-and-mim-and-what-s-the-difference.aspx.

[Mahmood 2012] Mahmood, Riyadh, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. "A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud." June 2012. http://cs.gmu.edu/~smalek/papers/AST2012.pdf.

[Maji 2010] Maji, Amiya Kumar, Kangli Hao, Salmin Sultana, and Saurabh Bagchi. "Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian." November 2010. https://engineering.purdue.edu/dcsl/publications/papers/2010/Android_issre10_submit.pdf.

[Maji 2012] Maji, Amiya K., Fahad A. Arshad, and Saurabh Bagchi. "An Empirical Study of the Robustness of Inter-component Communications in Android." June 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6263963&queryText%3DAn+Empirical+Study+of+the+Robustness+of+Inter-component+Communications+in+Android.

[Malek 2009] Malek, Sam, George Edwards, Yuriy Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav S. Sukhatme. "An architecture-driven software mobility framework." November 2009. http://robotics.usc.edu/publications/media/uploads/pubs/704.pdf.

[Malek 2010] Malek, Sam, George Edwards, Yuriy Brun, Hossein Tajalli, Joshua Garcia, Ivo Krka, Nenad Medvidovic, Marija Mikic-Rakic, and Gaurav Sukhatme. "An Architecture-Driven Software Mobility Framework." June 2010. http://cs.gmu.edu/~smalek/papers/JSS2010.pdf.

[Malek 2012] Malek, Sam, Naeem Esfahani, Thabet Kacem, Riyadh Mahmood, Nariman Mirzaei, and Angelos Stavrou. "A Framework for Automated Security Testing of Android Applications on the Cloud." June 2012. http://cs.gmu.edu/~smalek/papers/SERE2012.pdf.

[Manadhata 2008] Manadhata, Pratyusa K. An Attack Surface Metric. November 2008. CMU-CS-08-152. http://reports-archive.adm.cs.cmu.edu/anon/2008/CMU-CS-08-152.pdf

[Marforio] Marforio, Claudio, Aurelien Francilon, and Srdjan Capkun. *Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems*. ftp://ftp.inf.ethz.ch/doc/tech-reports/7xx/724.pdf.

[Marien 2016] Marien, John R. (Chair), Robert A. Martin (Co-Chair), February 2016, *How to put software assurance into contracts: An effort of the Department of Defense*

*Software Assurance (SwA) Community of Practice (CoP) Contract Language Working Group*.

[Matsudo 2012] Matsudo, Takayuki, Eiichiro Kodama, Jiahong Wang, and Toyoo Takata. "A Proposal of Security Advisory System at the Time of the Installation of Applications on Android OS." 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6354836&queryText%3D A+Proposal+of+Security+Advisory+System+at+the+Time+of+the+Installation+of+Appl ications+on+Android+OS%E2%80%9D%2C+Takayuki+Matsudo.

[Mattmann 2012] Mattmann, Chris A., Nenad Medvidovic, Sam Malek, George Edwards, and Somo Banerjee. "A Middleware Platform for Providing Mobile and Embedded Computing Instruction to Software Engineering Students." August 2012. http://cs.gmu.edu/~smalek/papers/TE2012.pdf.

[McDowell 2009] McDowell, Mindi and Allen Householder. "Security Tip (ST04-004), Understanding Firewalls." U.S. CERT. 2009. http://www.us-cert.gov/ncas/tips/ST04-004.

[McGraw 2011] McGraw, Gary. "Software Security." Cigital. 2011. http://www.cigital.com/papers/download/bsi1-swsec.pdf.

[McGraw, 2003] McGraw, Gary. Cigital, "How Now Software Security?" Cigital. 2003. http://www.cigital.com/whitepapers/dl/How_Now_Software_Security.pdf.

[McGraw 2006] McGraw, Gary. "Java Security for Smart Cards." Cigital. 2006. http://www.cigital.com/whitepapers/dl/Java_Security_for_Smart_Cards.pdf.

[Meier 2003] Meier, J.D., Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla and Anandha Murukan. Microsoft. 2003. "Improving Web Application Security: Threats and Countermeasures."

[Mello2015] Mello, John. "Highlights from the 2015 Gartner Magic Quadrant for application security testing." TechBeacon. 2015-08-27, http://techbeacon.com/highlights-2015-gartner-magic-quadrant-application-security-testing. This is a summary of the "2015 Gartner Magic Quadrant for application security testing."

[Microsoft Signature] Microsoft. "How to tell if a digital signature is trustworthy." http://office.microsoft.com/en-us/excel-help/how-to-tell-if-a-digital-signature-is-trustworthy-HA001230875.aspx

[Milano 2012] Milano, Diego Torrest. "Android Application Testing Guide." August 2012. http://ebookbrowsee.net/android-application-testing-guide-diego-torres-milano-pdf-d386205579.

[Mills 2012] Mills, Elinor. "Google now scanning Android apps for malware." *CNet*. February 2, 2012. http://news.cnet.com/8301-27080_3-57370650-245/google-now-scanning-android-apps-for-malware/.

[Miller 2007] Miller, Miller, Charlie and Zachary N. J. Peterson. March 1, 2007. *Analysis of Mutation and Generation-Based Fuzzing Whitepaper*. https://www.defcon.org/images/defcon-15/dc15-presentations/Miller/Whitepaper/dc-15-miller-WP.pdf or https://fuzzing.info/papers/.

[Mirzaei 2012] Mirzaei, Nariman, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. "Testing Android Apps through Symbolic Execution." November 2012. http://cs.gmu.edu/~smalek/papers/JPF2012.pdf.

[Motes 2011] Motes, LTC Gregory. "U.S. Army Mobile Application Development: A Coder's Perspective," February 2011. http://www.ausa.org/publications/armymagazine/archive/2011/2/Documents/FC_Motes_0211.pdf.

[Mobile App Testing Blog 2013] Mobile App Testing Blog. "Important Mobile App Privacy Recommendations." February 2013. http://www.mobileapptesting.com/important-mobile-app-privacy-recommendations/2013/02/.

[MobileMAN] MobileMAN. "GLOSSARY." http://mobileman.projects.supsi.ch/glossary.html.

[MobiForge] MobiForge Stats. https://mobiforge.com/stats

[Murmuria] Murmuria, Rahul, Jeffrey Medsger, Angelos Stavrou, and Jeffrey M. Voas. "Mobile Application and Device Power Usage Measurements." 2013. http://cs.gmu.edu/~astavrou/research/Android_Power_Measurements_Analysis_SERE_12.pdf.

[Muttik 2011] Muttik, Igor. "Securing Mobile Devices: Present and Future." 2011. http://www.mcafee.com/us/resources/reports/rp-securing-mobile-devices.pdf.

[NcNamee 2011] McNamee, Kevin (Kindsight). "Malware Analysis Report Trojan: AndroidOS / Droid Deluxe." September 2011. http://www.kindsight.net/sites/default/files/Kindsight_Malware_Analysis-Android-Trojan-DroidDeluxe-final_0.pdf.

[NDAA 2014] U.S. Congress. *National Defense Authorization Act for Fiscal Year 2014*. http://armedservices.house.gov/index.cfm/files/serve?File_id=215AC26C-A0E7-4B02-A63C-DD9D800AF2DB.

[NDIA 2008] National Defense Industrial Association (NDIA) System Assurance Committee. *Engineering for System Assurance*. August 2008. http://www.acq.osd.mil/se/docs/SA-Guidebook-v1-Oct2008.pdf.

[NIST SP 800-124rev1 2013] National Institute of Standards and Technology (NIST). "Guidelines for Managing the Security of Mobile Devices in the Enterprise." Sep 2013. http://csrc.nist.gov/publications/drafts/800-124r1/draft_sp800-124-rev1.pdf.

[NIST SP-163] National Institute of Standards and Technology (NIST). "Guidelines for Testing and Vetting Mobile Applications – DRAFT." 2013. http://www.nist.gov/itl/csd/mobile-device-security-meeting.cfm.

[NIST 2013] National Institute of Standards and Technology (NIST). "NIST Mobile Security & Forensics Page." 2013. http://csrc.nist.gov/groups/SNS/mobile_security/index.html.

[NIST 2012] National Institute of Standards and Technologies (NIST). "FIPS Special Publication 800-124 Revision 1 (Draft): Guidelines for Managing and Securing Mobile Devices in the Enterprise (Draft)." July 2012. http://csrc.nist.gov/publications/drafts/800-124r1/draft_sp800-124-rev1.pdf.

[NIST CFTT] National Institute for Standards and Technologies (NIST) Computer Forensics Tools Testing (CFTT). http://www.cftt.nist.gov/.

[NIST Ockham] NIST. "SATE V Ockham Sound Analysis Criteria." Retrieved July 2013. http://samate.nist.gov/SATE5OckhamCriteria.html.

[Northrup 2013] Northrup, Tony. "Firewalls." 2013. http://technet.microsoft.com/en-us/library/cc700820.aspx.

[NSA 2012] National Security Agency (NSA). "Security Configuration Recommendations for Apple iOS 5 Devices." March 2012. http://www.nsa.gov/ia/_files/os/applemac/Apple_iOS_5_Guide.pdf.

[NQ Mobile 2012] NQ Mobile. "Mobile Security Report: An In-Depth Look at Mobile Threats, Vulnerabilities, and Challenges." NQ Mobile. February 2012. http://docs.nq.com/2011_NQ_Mobile_Security_Report.pdf.

[NVD] National Vulnerability Database (NVD). "CWE - Common Weakness Enumeration." http://nvd.nist.gov/cwe.cfm.

[Open Group 2011] Open Group. Open Trusted Technology Provider Framework (O-TTPF). February 2011. https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=12341.

[Oracle Signature] Oracle. "Verifying a Digital Signature." 1995. http://docs.oracle.com/javase/tutorial/security/apisign/versig.html.

[OWASP] OWASP. "OWASP Mobile Security Project."
https://www.owasp.org/index.php/OWASP_Mobile_SecurityProject.

[Palmer 2001] Palmer, Gary. "A Road Map for Digital Forensic Research. Technical
Report DTR-T0010-01, DFRWS." November 2001.
http://isis.poly.edu/kulesh/forensics/docs/DFRWS_RM_Final.pdf.

[Patel] Patel, Parth. "Introducing ASEF – Android Security Evaluation Framework."
https://community.qualys.com/servlet/JiveServlet/downloadBody/3675-102-4-
6580/ASEF-Blog(4).pdf

[Pieterse 2012] Pieterse, Heloise, and Martin S. Olivier. "Android Botnets on the Rise:
Trends and Characteristics." August 2012.
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6320432&queryText%3D
Android+Botnets+on+the+Rise%3A+Trends+and+Characteristics.

[Pollock] Pollock, Clint. "The Mobile App Top 10 Risks."
https://www.owasp.org/images/9/94/MobileTopTen.pdf.

[Rausnitz 2013]. Rausnitz, Zach. "NIST vetting commercial Android apps for security,
battery use." *FierceMobileGovernment*. February 2013.
http://www.fiercemobilegovernment.com/story/nist-vetting-commercial-android-apps-
security-battery-use/2013-02-27.

[Rebel] Rebel Labs. "Code Quality Tools Review for 2013: Sonar, Findbugs, PMD and
Checkstyle: Catching up with Code Quality Tools from 2012." March 12, 2013.
http://zeroturnaround.com/rebellabs/code-quality-tools-review-for-2013-sonar-findbugs-
pmd-and-checkstyle/.

[Rhodes 2009] Rhodes, Boland, Fong, and Kass. NIST Interagency Report 7608,
"Software Assurance Using Structured Assurance Case Models." May 2009.
http://nvl.nist.gov/pub/nistpubs/ir/2009/ir7608.pdf.

[Riyadh] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam
Malek, Angelos Stavrou, "A Whitebox Approach for Automated Security Testing of
Android Applications on the Cloud." http://cs.gmu.edu/~smalek/papers/AST2012.pdf.

[SAFECode 2012] SAFECode (Editor Stacy Simpson and contributors Diego Baldini,
Gunter Bitz, David Dillard, Chris Fagan, Brad Minnis, and Dan Reddy). "Software
Integrity Controls: An Assurance-Based Approach to Minimizing Risks in the Software
Supply Chain." June 2010.
http://www.safecode.org/publications/SAFECode_Software_Integrity_Controls0610.pdf.

[Salter] Salter, Chris (NSA), O. Sami Saydjari (DARPA), Bruce Schneier, (Counterpane
Systems), Jim Wallner (NSA). "Toward A Secure System Engineering Methodology."
http://www.schneier.com/paper-secure-methodology.pdf.

[SCALe 2012] Robert C. Seacord, William Dormann, James McCurley, Philip Miller, Robert Stoddard, David Svoboda, Jefferson Welch (Source Code Analysis Laboratory (SCALe)). April 2012. http://www.sei.cmu.edu/library/abstracts/reports/12tn013.cfm.

[Schechter 2013] Schechter, Erik. "How-DARPA-delivers-tactical-apps-mobile-devices-field." December 2013. http://www.c4isrnet.com/article/M5/20131204/C4ISRNET13/312040021/How-DARPA-delivers-tactical-apps-mobile-devices-field.

[Seo 2012] Seo, Seung-Hyun, Dong-Guen Lee, and Kangbin Yim. "Analysis on Maliciousness for Mobile Applications." July 2012.

[Serebryany 2012] Serebryany, Konstantin, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." *Usenix Annual Technical Conference (ATC) 2012*. 2012. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker.

[Serebryan2012] Serebryan, Konstantin, Derek Bruening, Alexander Potapenko, and Dmitry Vyuko, "AddressSanitizer: A Fast Address Sanity Checker", *Proceedings of the USENIX Annual Technical Conference (ATC) 2012*. Boston, MA. http://research.google.com/pubs/pub37752.html

[Seriot] Seriot, Nicholas. "iPhone Privacy." http://seriot.ch/resources/talks_papers/iPhonePrivacy.pdf.

[Sethi 2011] Sethi, Amit, Omair Manzoor, and Tarun Sethi. "User Authentication on Mobile Devices". 2012. http://www.cigital.com/wp-content/uploads/downloads/2012/11/mobile-authentication.pdf.

[Shah] Shah, Kunjah. "Penetration Testing Android Applications." www.mcafee.com/us/resources/white-papers/foundstone/wp-pen-testing-Android-apps.pdf.

[Soltani 2013] Soltani, Ashkan and Timothy B. Lee. "Research shows how MacBook Webcams can spy on their users without warning." December 2013. http://www.washingtonpost.com/blogs/the-switch/wp/2013/12/18/research-shows-how-macbook-webcams-can-spy-on-their-users-without-warning/.

[Souppaya 2012] Souppaya, Murugiah, and Karen Scarfone. "Guidelines for Managing and Securing Mobile Devices in the Enterprise (Draft)." July 2012. http://csrc.nist.gov/publications/drafts/800-124r1/draft_sp800-124-rev1.pdf.

[Steele 2013] Steele, Colin. "Mobile device management versus mobile application management." 2013. http://searchconsumerization.techtarget.com/feature/Mobile-device-management-vs-mobile-application-management.

[Sutton2007] Sutton, Michael, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley. Upper Saddle River, NJ. 2007. ISBN 0-321-44611-9.

[SwAForum 2012] Software Assurance Forum. *Software Assurance in Acquisition and Contract Language*. May 18, 2012. https://buildsecurityin.us-cert.gov/swa/software-assurance-pocket-guide-series

[Szydlowski] Szydlowski, Martin, Manuel Egele, Christopher Kruegel, and Giovanni Vigna. "Challenges for Dynamic Analysis of iOS Applications." http://iseclab.org/papers/iphone-dynamic.pdf.

[Takanen2008] Takanen, Ari, Jared D. Demott, Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance.* Artech House. Boston, MA. 2008. ISBN 978-1-59693-214-2.

[Tesfay 2012] Tesfay, Werderufael Berhane, Todd Booth, and Karl Andersson. "Reputation Based Security Model for Android Applications." 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6296066.

[Thompson 1984] Thompson, Ken. "Reflections on Trusting Trust." Communications of the ACM. Volume 27, Number 8. pp. 761-763. April 1984. http://www.acm.org/classics/sep95.

[Tooke 2012] Tooke, William. "The Application Software Assurance Center of Excellence (ASACoE) Process." Department of Defense (DoD) Community of Practice meeting. September 19, 2012.

[Tracy 2012] Tracy, Kim W. "Mobile Application Development Experiences on Apple's iOS and Android OS." July 2012. http://www.deepdyve.com/lp/institute-of-electrical-and-electronics-engineers/mobile-application-development-experiences-on-apple-s-ios-and-android-vp3Z0XSIUw.

[Utest] Utest. "The Essential Guide to Mobile App Testing." http://www.utest.com/landing-blog/essential-guide-mobile-app testing?ls=Banner%20Ad&mc=Display-Blog_MAT-MenuBar

[viaForensics] viaForensics. "Mobile Security Risk Report: Understanding the Security Impact of iOS and Android in the Enterprise." https://viaforensics.com/resources/reports/mobile-security-risk-report/.

[ViaForensics 2011] ViaForensics. "appWatchdog Findings: Sensitive User Data Stored on Android and iPhone Devices." July 2011. https://viaforensics.com/resources/reports/mobile-app-security-study/overall/.

[Veracode 2011] Veracode. "State of Software Security Report: The Intractable Problem of Insecure Software Volume 4." December 7, 2011. http://media.blackhat.com/bh-eu-12/Wysopal/bh-eu-12-Wysopal-State_of_Software_Security-WP.pdf.

[VMWare and parasoft 2012] VMWare and Parasoft. "Mobile Powered Government: Driving Increasingly Productive, Efficient Agencies." February 2012. http://www.meritalk.com/pdfs/Mobile_Powered_Government_Media_Coverage.pdf.

[Voas 2013] Voas, Jeffrey, Steve Quirolgico, Christoph Michael, Irena Bojanova, and Karen Scarfone. "Technical Considerations for Vetting Applications for Android Mobile Devices (Draft). NIST Special Publication (SP) 800-163." November 2013.

[Williams2012] Williams, Jeff, and Arshan Dabirsiaghi. "The Unfortunate Reality of Insecure Libraries." March 2012. https://www.aspectsecurity.com/uploads/downloads/2012/03/Aspect-Security-The-Unfortunate-Reality-of-Insecure-Libraries.pdf.

[Walker 2013] Walker. "DHS to stand up 'car wash' for mobile apps." *FierceMobileGovernment*. July 2013. http://www.fiercemobilegovernment.com/story/dhs-stand-car-wash-mobile-apps/2013-07-17.

[Wang 2011] Wang, Zhaohui and Angelos Stavrou. "Exploiting Smart-Phone USB Connectivity for Fun and Profit." *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC).* 2011. http://cs.gmu.edu/~astavrou/research/acsac10.pdf or http://www.acsac.org/2010/preview/2010-acsac-proceedings.pdf.

[Wang, 2012] Wang, Zhaohui, Ryan Johnson, and Angelos Stavrou. "Attestation & Authentication for USB Communications." 2012. http://cs.gmu.edu/~astavrou/research/USB_Attestation_SERE_2012.pdf.

[Weber] Weber, Sam, Paul A. Karger, and Amit Paradkar. "A Software Flaw Taxonomy: Aiming Tools At Security." http://cwe.mitre.org/documents/sources/ASoftwareFlawTaxonomy-AimingToolsatSecurity%5BWeber,Karger,Paradkar%5D.pdf.

[Wei 2012] Wei, Xuetao, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. "Malicious Android Applications in the Enterprise: What Do They Do and How Do We Fix It?" http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6313688&queryText%3D

[Wheeler 1996] Wheeler, David A., Bill Brykczynski, and Reginald N. Meeson, Jr. *Software Inspection: An Industry Best Practice*. IEEE Computer Society Press. 1996.

[Wheeler2009] Wheeler, David A. *Fully Countering Trusting Trust through Diverse Double-Compiling*. 2009. http://www.dwheeler.com/trusting-trust/.

[Wheeler 2012] Wheeler, David A. and Rama S. Moorthy. *SOAR-Lite Phase I*. August 2012.

[Wheeler2015] Wheeler, David A. *How to Prevent the next Heartbleed*. 2015-09-18. Retrieved 2016-03-07. http://www.dwheeler.com/essays/heartbleed.html

[Wheeler2016] Wheeler, David A. *The Apple goto fail vulnerability: lessons learned*. 2016-03-01. Retrieved 2016-10-27. http://www.dwheeler.com/essays/apple-goto-fail.html

[Wei 2012] Wei, Xuetao, Lorenzo Gomez, Iulian Neamtiu, Michalis Faloutsos. "Permission Evolution in the Android Ecosystem." 2012. http://www.cs.ucr.edu/~neamtiu/pubs/acsac12wei.pdf.

[Wei 2012] Wei, Xuetao, Lorenzo Gomez, Iulian Neamtiu, Michalis Faloutsos. "ProfileDroid: Multi-layer Profiling of Android Applications." 2012. http://www.cs.ucr.edu/~neamtiu/pubs/mobicom12wei.pdf.

[Whitwam 2011] Whitwam, Ryan. "Android Security Threats and How You Can Stay Safe." September 2011. http://www.extremetech.com/mobile/95147-Android-security-threats-and-what-users-can-do-to-stay-safe.

[Woody2014] Woody, Carol, Robert Eillison, and William Nichols. Predicting Software Assurance Using Quality and Reliability Measures. December 2014. Technical Note CMU/SEI-2014-TN-026. Carnegie Mellon University (GMU) Software Engineering Institute (SEI) CERT Division/SSD.

[Wu, 2012] Wu, Dong-Jie, Te-En Wei, and Kuo-Ping Wu. "DroidMat: Android Malware Detection through Manifest and API Calls Tracing." *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security (Asia JCIS)*. August 2012. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6298136&queryText%3DDroidMat%3A+Android+Malware+Detection+through+Manifest+and+API+Calls+Tracing.

[Xu 2013] Xu, Wei, Fangfang Zhang, and Sencun Zhu. "Permlyzer: Analyzing Permission Usage." *Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) 2013*. http://www.cse.psu.edu/~szhu/papers/permlyzer.pdf.

[Zalewki2014] Zalewski, Michał (aka lcamtuf), "Pulling JPEGs out of thin air," *lcamtuf's blog*, November 07, 2014, http://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html, retrieved March 7, 2016.

[Zhang 2013] Zhang, Yulong, Hui Xue, Tao Wei, Dawn Song.  "Monitoring Vulnaggressive Apps on Google Play."  *FireEye Blog*.  November 2013. http://www.fireeye.com/blog/technical/2013/11/monitoring-vulnaggressive-apps-on-google-play.html.

[Zhou 2012] Zhou, Yajin, Xuxian Jiang.  "Dissecting Android Malware: Characterization and Evolution," *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*.  May 2012. http://www.csc.ncsu.edu/faculty/jiang/pubs/OAKLAND12.pdf.

| 1. REPORT DATE (DD-MM-YY)<br>00-11-16 | 2. REPORT TYPE<br>Final | | 3. DATES COVERED (From – To) | | |
|---|---|---|---|---|---|
| 4. TITLE AND SUBTITLE<br><br>State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016 v.2 | | | 5a. CONTRACT NUMBER<br>   HQ0034-14-D-0001 | | |
| | | | 5b. GRANT NUMBER | | |
| | | | 5c. PROGRAM ELEMENT NUMBERS | | |
| 6. AUTHOR(S)<br><br>   David A. Wheeler, Amy E. Henninger | | | 5d. PROJECT NUMBER<br>AU-5-3856 | | |
| | | | 5e. TASK NUMBER | | |
| | | | 5f. WORK UNIT NUMBER | | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES<br><br>Institute for Defense Analyses<br>4850 Mark Center Drive<br>Alexandria, VA 22311-1882 | | | 8. PERFORMING ORGANIZATION<br>REPORT NUMBER<br>P-8005<br>H 2016-000598 | | |
| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br><br>Kristen Baldwin, Acting Deputy Secretary of Defense, Systems Engineering<br>Office of the Deputy Assistant Secretary of Defense for Systems Engineering; Acquisition Technololgy and Logistics<br>3030 Defense Pentagon, Room 3C167, Washington, DC 20301-3030 | | | 10. SPONSOR'S / MONITOR'S ACRONYM<br>DASD-SE | | |
| | | | 11. SPONSOR'S / MONITOR'S REPORT<br>NUMBER(S) | | |
| 12. DISTRIBUTION / AVAILABILITY STATEMENT<br><br>Approved for public release; distribution is unlimited. | | | | | |
| 13. SUPPLEMENTARY NOTES<br><br>Project Leader:  E. Kenneth Hong Fong | | | | | |
| 14. ABSTRACT<br><br>Unintentional and intentionally inserted vulnerabilities in software can provide adversaries with various avenues to reduce system effectiveness, render systems useless, or even use our systems against us.  Unfortunately, it can be difficult to determine what types of tools and techniques exist for evaluating software, and where their use is appropriate.<br>This paper is written to enable DoD program managers (PMs), and their staff, to make effective software assurance and software supply chain risk management (SCRM) decisions, particularly when they are developing and executing their program protection plan (PPP). A secondary purpose is to inform DoD policymakers who are developing software policies.  This paper describes an overall process for selecting and using appropriate analysis tool/technique types for evaluating software: (1) Select technical objectives based on context; (2) Select tool/technique types to address those technical objectives; (3) Select tools/techniques; (4) Summarize selection as part of a Program Protection Plan (PPP); (5) Apply the tools/techniques and report the results.  This paper identifies 59 types of tools and techniques available for analyzing software, along with a mapping between these tool/technique types and technical objectives, to help readers identify and select types of tools and techniques. | | | | | |
| 15. SUBJECT TERMS<br><br>Software assurance, software vulnerabilities, software security assurance, supply chain risk management, SCRM, DoD, tools, static analysis, dynamic analysis, hybrid analysis, weakness analysis, static analysis tools, dynamic analysis tools, unintentional, intentional, technical objectives, program protection plan, PPP, target of evaluation, TOE, state-of-the-art, SOAR | | | | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT<br>Unlimited | 18. NUMBER OF PAGES<br>268 | 19a. NAME OF RESPONSIBLE PERSON<br>Kristen Baldwin, Acting Deputy Secretary of Defense, Systems Engineering |
|---|---|---|---|---|---|
| a. REPORT<br>Unclassified | b. ABSTRACT<br>Unclassified | c. THIS PAGE<br>Unclassified | | | 19b. TELEPHONE NUMBER (Include Area Code)<br>703-695-7417 |