INSTITUTE FOR DEFENSE ANALYSES

Securely Using Software Assurance (SwA) Tools in the Software Development Environment

E. Kenneth Hong Fong, *Project Leader*

David A. Wheeler
Daniel J. Reddy

July 2018

IDA Document
P-9166

INSTITUTE FOR DEFENSE ANALYSES
4850 Mark Center Drive
Alexandria, Virginia 22311-1882

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.

About This Publication

This work was conducted by the Institute for Defense Analyses (IDA) under contract HQ0034-14-D-0001, Task AU-5-3856, "Enhancing Program Protection Through Effective Systems Assurance," for OUSD(R&E) Enterprise Engineering. The views, opinions, and findings should not be construed as representing the official position of either the Department of Defense or the sponsoring organization.

Acknowledgments

Reginald N. Meeson

For more information:

E. Kenneth Hong Fong, Project Leader
ehongfon@ida.org, 703-578-2753

Margaret E. Myers, Director, Information Technology and Systems Division
mmyers@ida.org, 703-578-2782

INSTITUTE FOR DEFENSE ANALYSES

IDA Document P-9166

Securely Using Software Assurance (SwA) Tools in the Software Development Environment

E. Kenneth Hong Fong, *Project Leader*

David A. Wheeler
Daniel J. Reddy

Executive Summary

Software assurance (SwA) may be defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner" [CNSSI 4009]. Since modern systems are under constant attack, sufficient SwA is vital. In practice, a suite of SwA tools is necessary to help achieve this, as previously shown in the IDA paper *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation 2016* [Wheeler 2016].

However, there are potential challenges in securely using a suite of SwA tools.

Software development environments (SDEs) are increasingly under focused attack, since subverting software during development can be easier than subverting it after it is deployed. One mechanism for subverting SDEs is to exploit vulnerabilities in an SDE's tools or to provide maliciously subverted tools to an SDE. Adversaries know that a suite of SwA tools is necessary for higher-assurance software and therefore may increasingly try to attack our systems through our SwA tools.

In addition, SwA tools have properties that can make them attractive targets for adversaries. Projects should use a large set of SwA tools to increase the probability of detecting vulnerabilities. In addition, SwA tools often have privileged access to information such as source code and test data. However, a large number of tools with privileged access increases the attack surface (the possible points of attack), and defenders will have difficulty reviewing all those SwA tools before use. In addition, updates of SwA tools are typically deployed rapidly, adding to the difficulty of timely review.

Many organizations (including many parts of the Department of Defense (DoD)) impose significant restrictions on installing and updating any software (including SwA tools), especially in privileged environments or on an organization's main network. There are valid reasons for these restrictions, as they are one way to counter some risks. However, these restrictions can make it especially difficult to use a full suite of SwA tools, even though a suite is necessary to achieve adequate SwA in practice.

The goal of this paper is to help ease the deployment of SwA tools, by countering potential objections to using them. To achieve this, we discuss how to protect against potential supply chain risks of SwA tools themselves, including how to protect the SDE in general against supply chain risks, and how the mechanisms to counter SwA tool risks fit into the SDE. We show that it is possible to modify SDE practices to use a wide variety of SwA tools and still manage the inherent risks. Isolation mechanisms can be

used, for example, to separate tools and restrict access for specific tasks. Isolation mechanisms can be automated and may reduce risk in a relatively uncomplicated manner.

We first discuss past attacks and countermeasures (Chapter 2), the problem including the overall security goals (Chapter 3), and security weaknesses of some common approaches (Chapter 4). This is followed by how to develop solutions (Chapter 5), sample solutions (Chapter 6), and conclusions (Chapter 7). Appendices provide proofs of concept (Appendix A), a quick implementation guide for the medium solution (Appendix B), a discussion of DoD policies (Appendix C), and a list of acronyms and abbreviations (Appendix D) Together, this material discusses how to protect against the supply chain risks that could be caused by SwA tools themselves and to help ease the deployment of the many SwA tools necessary for higher-assurance software.

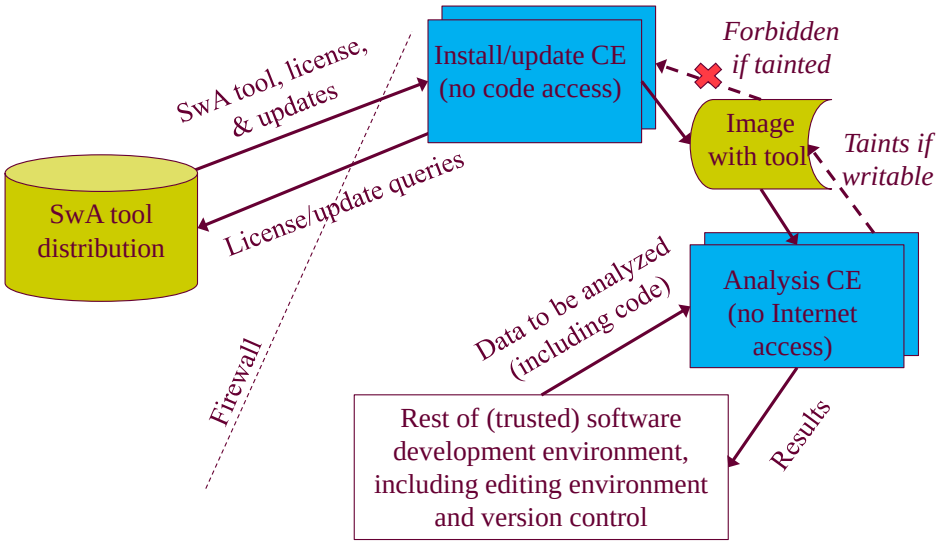


Figure 1. Medium Protection Illustration

In particular, the proposed "medium protection" approach discussed in detail in Chapter 6 should be easy to incorporate in existing SDEs (this approach is briefly summarized in Appendix B and illustrated in Figure 1). This sample solution reduces risks by using isolation mechanisms to separate environments based on the task to be done (install/update and analysis). This sample solution can be automated, and in some circumstances it may reduce risk in a relatively uncomplicated manner. These automations could be implemented with simple scripts that are shared widely, making the approach easy to implement.

We recommend that organizations fully embrace the use of many SwA tools when developing software. Where appropriate, they should consider taking the additional steps discussed here if they determine that the risks of using SwA tools are otherwise too high. Our hope is that this information will lead to the widespread safe use of suites of SwA tools.

Contents

Figures

Tables

1. Introduction

Software assurance (SwA) may be defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner" [CNSSI 4009]. Like safety, this is an idealized goal, and achieving sufficient SwA requires a combination of measures to remove vulnerabilities and their impact to systems. Since modern systems are under constant attack, sufficient SwA is vital.

In practice, there is a clear need for "predictable and scalable analysis tools that increase trust in software" [Wagoner]. Indeed, a suite of SwA tools is necessary to detect vulnerabilities adequately enough to achieve good SwA. Different SwA tools find different things, and few guarantee that they will find everything [Wheeler 2016]. The Building Security in Maturity Model (BSIMM) is the result of a multiyear study of real-world software security initiatives, and version 8 of the BSIMM survey shows that many different kinds of SwA tools are in use today across a variety of organizations [BSIMM 2017].[1] A step-by-step discussion of how to use SwA static analysis tools is provided in [Kupsch 2016]. Using many different kinds of SwA tools is an excellent approach for detecting vulnerabilities in software during development. However, there are challenges in installing many different kinds of SwA tools.

Many organizations (including many parts of the DoD) impose significant restrictions on installing and updating software (including SwA tools), especially in privileged environments or an organization's main network. Examples of such restrictions and review processes include the Army Certificate of Networthiness (CoN) [ArmyCoN], Department of the Navy's Application and Database Management System (DADMS), The Defense Information Systems Agency's DoD Information Network (DODIN) Approved Products List (APL), the Risk Management Framework (RMF) "Assess Only" approach, and Common Criteria evaluations. These processes can take significant time and money, especially if there is a perception of increased risk. These delays and costs can make it difficult to deploy many SwA tools.

[1] BSIMM8 identifies several activities that involve tools, including CR1.4 (Use automated tools along with manual review, aka static source code vulnerability analysis), CR3.4 (Automate malicious code detection), ST2.1 (Integrate black-box security tools into the QA process, which focuses on dynamic analysis approaches like web application scanning), CR3.1 (Use automated tools with tailored rules), SR2.4 (Identify open source [with known vulnerabilities], aka origin analysis). Note that there are many different kinds of tools identified.

A software development environment (SDE) is the collection of hardware and software tools a system developer uses to build software systems; it augments or automates the activities comprising the software development cycle, including tasks such as configuration management, project management, and team management [Dart 1992]. Alternative terms for SDE include *software engineering environment* (SEE). Note that an SDE is a collection of tools and supporting processes, not any one tool. The term *integrated development environment* (IDE) is usually used to describe a particular type of tool (an editor and a few related tools) used in an SDE; an IDE is not an SDE by itself. Similarly, the term *programming environment* often refers to just the part of the SDE involved directly in implementation. The term *SDE* is used even if the software is in a sustainment (maintenance) stage/phase. Modern software development takes place in some kind of SDE.

Whether or not an organization imposes significant restrictions on installing and updating software, there is a disturbing trend: SDEs themselves are increasingly under focused attack. Examples of attacks on SDEs include XCodeGhost, Expensive Wall / Shady SDK, and others, as described more fully in Chapter 2. For example, XCodeGhost distributed subverted software development tools that led to the deployment of subverted systems. Thus, the risk of attacks on SDEs, including via software tools, should not be ignored. In addition, adversaries know that a suite of SwA tools is necessary for higher-assurance software and therefore may increasingly try to attack our systems through our SwA tools (which may be unintentionally vulnerable or subverted).

In this paper, we focus on identifying practical approaches to protect against the supply chain risks of SwA tools themselves, with some discussion about how to protect the SDE in general against supply chain risks and how the mechanisms to counter SwA tool risks fit into them. Our goal is to help ease the deployment of SwA tools by countering a potential objection to using them.

This focus on supply chain risks of SwA tools may seem surprising, since SwA tools are specifically selected and used to improve the security posture of software under development and/or its execution environment. However, SwA tools can also pose a risk:

1. Like many other tools in the SDE, SwA tools often have privileged access to information such as source code and test data as typically deployed today.
2. SwA tools are used differently in the SDE in ways that make them easier vectors for attackers to exploit:
    a. Projects should use a large set of SwA tools to increase the probability of finding vulnerabilities. The number of tools, however, provides attackers a larger attack surface (the set of all SwA tools used by a project), and defenders will have difficulty reviewing all those tools before use. Projects will typically use a large set of SwA tools so that they can address a range of

technical objectives, because most SwA tools can only address a few technical objectives. In contrast, in most cases, only one or a few tools are used for any particular purpose in an SDE (e.g., compilers).

b. Updates of SwA tools are typically deployed rapidly. This makes fully reviewing SwA tools in a timely way even more difficult. This is in contrast to tools like compilers, which, in many environments, are updated only after careful review (since any update could create a new subtle error).

An SwA tool can be a risk even if an SwA tool supplier is not malicious. A determined adversary could create a malicious SwA tool, but an adversary could also attempt to subvert a non-malicious supplier's SDE or supply chain (either upstream or downstream). The unintentional vulnerabilities in an SwA tool might also be exploited and allow a foothold for further attack. This means that, in some cases, the risks of SwA tools should be addressed to ease deployment of many SwA tools.

In this paper, we identify practical approaches to protect against the supply chain risks of SwA tools themselves, so that projects can use and update a large number of SwA tools with much greater confidence and acceptable risk. These approaches are primarily technical measures, so that they can be automated. If these technical measures can be relatively low cost and scale up to large systems they will reduce a barrier to using SwA tools. More SwA tools, updated more often, should enable projects to produce more secure software.

There are other approaches to deal with these risks without using scalable technical measures as described in this paper. However, those alternatives will often be worse. One approach is to simply ignore the risk and blindly accept the consequences, but this is obviously not a good approach.

A second approach would be to use relatively few SwA tools. Unfortunately, that could lead to more undetected vulnerabilities left within the software under development.

A third approach would be to review SwA tools and their suppliers before using these tools. Reviews can be helpful in reducing risk. There are two major kinds of reviews, and each kind has limitations:

- *Acquirers can review suppliers and/or their goods and services*. Review of SwA tools for risks is a good idea, and it can be valuable to evaluate SwA tool suppliers for the potential risks they themselves pose. However, independent reviews of potential suppliers can be difficult and expensive, and there is always the potential for missing important problems.
- *Potential suppliers can present evidence to acquirers of adherence to best practices.* For example, the Open Group's Open Trusted Technology Provider Standard (O-TTPS), aka ISO/IEC 20243, is an "open standard containing a set of

organizational guidelines, requirements, and recommendations for integrators, providers, and component suppliers to enhance the security of the global supply chain and the integrity of commercial off-the-shelf (COTS) information and communication technology (ICT)." [OpenGroup 2014]. Open source software (OSS) can attempt to achieve a badge through the Linux Foundation Core Infrastructure Initiative (CII) Best Practices project [LF2017]. However, in many cases, suppliers do not have such evidence to provide (e.g., they might not be following any particular best practices, or may refuse to provide evidence of it).

Reviewing a large number of SwA tools, however, can be challenging. As noted above, it is best to use a wide variety of SwA tools, since different tools can help find different problems. In addition, it's valuable to use the latest versions of SwA tools, since they are often rapidly improved. Reviews are still valuable for SwA tools and their suppliers, especially if either are high risk. However, the large number of tools and their rapid updates make reviews (by acquirer or supplier) difficult to apply to all SwA tools, and it is wise to be skeptical even after reviews. Other measures, such as those described in this paper, can provide additional forms of protection.

A fourth approach is to disable all Internet access to SDEs. This can prevent many kinds of exfiltration. However, analysis results must eventually come out of SwA tools and SDEs, so this still provides opportunities for exfiltration. In addition, restricting exfiltration paths does not counter malicious insertion of code. What's more, isolated SDEs can be unnecessarily hard to work within. For example, tool updates may be extremely difficult, and yet these updates can increase overall SwA.

This paper is written in support of the efforts of the DoD Joint Federated Assurance Center (JFAC). The JFAC charter section 4 states that the "JFAC is the federation of all [DoD] entities having software and hardware assurance capabilities needed by programs. The JFAC will develop, maintain, and offer software and hardware vulnerability detection, analysis, and remediation capabilities…" [JFAC 2015]. In particular, the JFAC is to provide guidance and best practices (charter section (4)(a)), establish and enable efficient and affordable acquisition and use of SwA tools (charter section (4)(d)), and coordinate access to and capability for applying tools and support environments (charter section (5)(a)(3)). Again, our goal is to help ease the deployment of SwA tools by countering a potential objection to using them.

We first discuss past attacks and countermeasures (Chapter 2), the problem including the overall security goals (Chapter 3), and security weaknesses of some common approaches (Chapter 4). This is followed by how to develop solutions (Chapter 5), sample solutions (Chapter 6), and conclusions (Chapter 7). Appendices provide proofs of concept (Appendix A), a quick implementation guide for the medium solution (Appendix B), a discussion of DoD policies (Appendix C), and a list of acronyms and abbreviations (Appendix D). Together, this material discusses how to protect against the

supply chain risks that could be caused by SwA tools themselves and to help ease the deployment of the many SwA tools necessary for higher-assurance software.

2. Past Attacks and Countermeasures

This chapter discusses past attacks involving the software supply chain or SDEs, as well as past and current methods of countering such attacks. For more about Department of Defense policies related to countering attacks on the supply chain or SDEs, see Appendix C.

A. Past Attacks Involving the Software Supply Chain or SDEs

Many past attacks have involved the software supply chain or SDE, typically by exploiting the supply chain to deliver attacks. These attacks appear to be accelerating in frequency and effectiveness. Table 2-1 lists some prominent software supply chain attacks. Attack notes marked with "*" are also in the "Software Supply Chain Attacks" placemat [Shaw 2017a]. That placemat's supporting reference list [Shaw 2017b] provides additional notes in those cases; supporting sources for each attack are provided in the text following the table.

Table 2-1. Some Prominent Software Supply Chain or SDE-based Attacks

Date	Attack Name	Target Technology	Attack Vector	Attack Note
Nov 2003	Unnamed attack on Linux kernel	Linux kernel source code	Version control system	Unknown attacker attempted to insert a vulnerability into the Linux kernel
Mar 2011	RSA	RSA SecurID	Targeted cyber attack	Targeted cyber attack on RSA systems led to an attempted though failed attack on a DoD contractor (an RSA customer)
Oct 2014	Unnamed attack via Tor	Downloaded Windows software	Tor	Software downloaded via anonymizing service Tor was subverted
Dec 2015	XCodeGhost	iOS	SDE tool attack	"Fake version of the developer tool distributed to site frequented by developers"*
Jan 2017	Expensive Wall / Shady SDK	Android	SDE tool attack	"Obfuscation used by malware developers to encrypted malicious code, allowing evasion of anti-malware protections"*
Mar 2017	Dimnie	GitHub users	Email + Malicious Word document + PowerShell	Malicious Trojan designed to steal passwords, download sensitive files, etc. from developers using the widely used GitHub service

Date	Attack Name	Target Technology	Attack Vector	Attack Note
Jun 2017	PyPI attack	Python	Repository attack	False libraries uploaded to the Python Package Index (PyPI) repository, similar to a typosquatting attack*
Jun 2017	NotPetya	MeDoc	Patch site attack	"Software infrastructure compromise to tamper with code."*
Jul 2017	Shadowpad	Network management software suite	Source code attack	"Backdoor injected into a network management software suite then pushed through software update"*
Aug 2017	Floxif	CCleaner	Insider/download site attack	"Infiltration into development or distribution process before cryptographic signature for software occurred"*
Aug 2017	Chrome extension attacks	Chrome extensions	Phishing on developers of extensions	Developers of Chrome extensions had their login credentials stolen through a phishing attack; attackers modified the extensions and compromised 4.8M users
Aug 2017	HackTask	JavaScript	Software development tool attack	Typosquatting attack*
Oct 2017	North Korean attack on Hauri	Hauri anti-virus software	Source code attack	"Infiltrated network of a company providing computer anti-virus service [for South Korean military]"*

In November 2003, someone attempted to modify the Linux kernel to insert a vulnerability. This attack was detected by the version control (VC) system, as well as by developers and source code conventions, and was never delivered to users [Andrews 2003].

RSA (then the security division of EMC; now part of Dell Technologies) publicly disclosed on March 17, 2011, that they had detected a targeted cyber attack on their systems and that certain information related to their RSA SecurID product had been extracted. [Coviello 2011]. One RSA customer, "Lockheed Martin, has confirmed that information taken from RSA has been used as part of a broader attack against it; an attack that the customer successfully thwarted." [EMC 2011]. The RSA executive chairman noted, in testimony to a committee of the U.S. House of Representatives, that "we are seeing increases in attacks on one organization to be leveraged in an attack on another organization" [Coviello 2011].

In 2014, it was revealed that Windows software downloaded via Tor (an anonymizing service) in some cases was modified to include malicious code. Even files downloaded through Windows update could be affected [Hern 2014].

The XCodeGhost attack released a subverted version of "Apple's legitimate iOS/OSX app development tool called Xcode to distribute [malicious] code in legitimate apps. XcodeGhost's creators repackaged Xcode installers with the malicious code and published links to the installer on many popular forums for iOS/OS X developers. Developers were enticed into downloading this tampered version of Xcode because it would download much faster in China than the official version of Xcode from Apple's Mac App Store. When the developers installed what they thought was a safe Apple dev tool, they actually got a tampered version that would compile the malicious code alongside their actual app's code. These developers, unaware that their apps had been tampered with, then submitted those apps to the App Store for distribution to iOS devices" [Cockerill 2015]. XcodeGhost is the first compiler malware in OS X [Xiao 2015a] and thus is an example of a subverted software development tool. The list of infected apps includes some of the most popular apps in China, including the ride-hailing app Didi Kuaidi and WeChat [Goodin 2015], and thus affected hundreds of millions of users [Xiao 2015b].

"Malware authors hid malicious code inside a software development kit (SDK) that developers embedded in their Android apps, unwittingly exposing their users to a mobile malware strain that Check Point identifies as ExpensiveWall… Check Point says it found the malware hiding in over 100 apps uploaded on the official Google Play Store… the malicious apps were downloaded between 5.9 million and 21.1 million times" [Cimpanu 2017-ExpensiveWall].

"Open source developers who use GitHub are in the cross-hairs of advanced malware that can steal passwords, download sensitive files, take screenshots, and self-destruct when necessary. Dimnie, as the reconnaissance and espionage Trojan is known, has largely flown under the radar for the past three years. It mostly targeted Russians until early this year, when a new campaign took aim at multiple owners of GitHub repositories… The campaign targeting GitHub users starts with e-mails that attach a booby-trapped Microsoft Word document. The file contains a malicious macro that uses PowerShell commands to download and execute the payloads… It's not hard to come up with plausible theories why either nation-sponsored or financially motivated hackers would want to spy on this demographic. What's clear now is that someone is devoting considerable time and expertise to make that happen." [Goodin2017-GitHub]

The PyPI attack of 2017 occurred when developers unknowingly used malicious modules. The Python language provides many built-in libraries, as well as a way to download many other libraries. Attackers created malicious libraries with the names of built-in libraries, and unknowing developers downloaded the malicious ones instead.

[Goodin 2017PyPI]. This was similar to a typosquatting attack discussed in a 2016 research paper, which showed that "typosquatting" (creating packages with names similar to popular packages) could lead to execution of potentially subverted code; even some military sites ended up running the potentially subverted code. [Tschacher 2016] [Goodin 2017Pypi]

The NotPetya attack of 2017 began with the penetration of the network of "the small Ukrainian software firm MeDoc, which sells a piece of accounting software that's used by roughly 80% of Ukrainian businesses. By injecting a tweaked version of a file into updates of the software, [the attackers] were able to start spreading backdoored versions of MeDoc software as early as April of this year that were then used in late June to inject the ransomware known [as NotPetya or Nyetya] that spread through victims' networks from that initial MeDoc entrypoint. … But just as disturbing … is the continuing threat it represents: that innocent software updates could be used to silently spread malware. … One reason [attackers] are turning to software updates as an inroad into vulnerable computers may be the growing use of 'whitelisting' as a security measure, says Matthew Green, a security-focused computer science professor at John Hopkins University. Whitelisting strictly limits what can be installed on a computer to only approved programs, forcing resourceful [attackers] to hijack those whitelisted programs rather than install their own. 'As weak points get closed up on the company side, they'll go after suppliers,' says Green. 'We don't have many defenses against this. When you download an application, you trust it.'… Even if the company had carefully signed its code, Green points out… it likely wouldn't have protected the victims in the MeDoc case. … [attackers] were deep enough in MeDoc's network that they likely could have stolen the cryptographic key and signed the malicious update themselves." [Greenberg 2017-Petya] Kaspersky labs named this 2017 variant NotPetya, as it is related but has significant differences from earlier 2016 malicious software named Petya [BBC 2017]. NotPetya used a variety of techniques to spread to other computers, including the ExternalBlue and EternalRomance exploits purportedly developed by the U.S. National Security Agency (NSA) [Fruhlinger 2017].

"ShadowPad is one of the largest known supply-chain attacks… [NetSarang is] server management software produced by a legitimate company and used by hundreds of customers in industries like financial services, education, telecoms, manufacturing, energy, and transportation. [The latest version was making suspicious requests, and] the vendor did not mean for the software to make these requests... the suspicious requests were [caused by] a malicious module hidden inside a recent version of the legitimate software. Following the installation of an infected software update, the malicious module would start sending DNS-queries to specific domains (its command and control server)… If the attackers considered the system to be 'interesting', the command server would reply and activate a fully fledged backdoor platform that would silently deploy itself

inside the attacked computer. After that, on command from the attackers, the backdoor platform would be able to download and execute further malicious code." Kaspersky's Igor Soumenkov said, "ShadowPad is an example of how dangerous and wide-scale a successful supply-chain attack can be… given the opportunities for reach and data collection it gives to the attackers, most likely it will be reproduced again and again with some other widely used software component." [Kaspersky 2017]

CCleaner is an example of an attack on the software development/distribution system. "Avast cryptographically signs installations and updates for CCleaner, so that no imposter can spoof its downloads [but attackers] infiltrated Avast's software development or distribution process before that signature occurred, so that the antivirus firm was essentially putting its stamp of approval on malware, and pushing it out to consumers." The attack was unnoticed for almost a month [Cimpanu 2017-CCleaner]. The Greenberg article made the general observation that attackers are increasingly exploiting "the digital supply chain to plant tainted code that hides in software companies' own systems of installation and updates, hijacking those trusted channels to stealthily spread their malicious code." [Greenberg 2017]

Developers of Chrome extensions had their login credentials stolen through a phishing attack. Attackers modified the extensions and compromised 4.8M users [Maunder 2017].

An attacker with the account name "HackTask" uploaded at least 38 malicious packages to npm (the widely used JavaScript package repository). "The attacker used a technique called 'typo-squatting' to register packages with names similar to popular libraries, but containing typos in their names. For example, the attacker registered a malicious package named 'mongose' [sic] that contained the source of the legitimate Mongoose project plus extra malicious code. The malicious code in this projects would execute when developers would compile and run their… JavaScript projects. The code would collect local environment variables and upload them to the attacker's server… The attack is dangerous because some information such as hard-coded passwords or API access tokens is stored as environment variables." [Cimpanu 2017-JavaScript]

North Korea "reportedly infiltrated Hauri, a South Korean company that provides antivirus software to that country's military [and North Korea was] able to grab classified data that included joint US-South Korea planning in event of war." [Barrett 2017]

These examples of real-world attacks indicate that there is a real need to protect against software supply chain and SDE attacks that impact developers.

B. Past and Current Methods of Countering Attacks on the Software Supply Chain or SDEs

The need to address the software supply chain, and to design an SDE to address development and sustainment needs, is not a new one. The SAFECode guide to software integrity controls [Simpson 2010] looks at software supply chain security, including upstream and downstream of the SDE as well as its impact on the SDE itself. Sutherland [1989] discusses "nurturing of a systems development environment to precisely meet the needs of a company." The Trusted Software Development Methodology (TSDM), later named the Trusted Software Methodology (TSM), discusses various approaches to implement environmental administration and controls to counter unintentional and intentional attacks on the SDE itself [GE 1991].

Many papers discuss how to create an SDE, or at least a collection of tools and processes, to help develop safe and/or secure software (e.g., [Hussein 2017]). Many documents discuss how to develop secure software (e.g., [SAFECode 2018] [Wheeler 2015-programming]). However, these topics are not the focus of this paper. Instead, this paper focuses on protecting the SDE itself, particularly on running SwA tools safely.

There are processes and services that evaluate products, processes, and/or people to help address supply chain risk management (SCRM). Evaluation processes such as the Common Criteria are designed to evaluate products. Guidance for addressing SCRM in DoD systems, including a list of potential key practices, is provided in [Wheeler 2010]. Other approaches are designed to measure organizations (e.g., BitSight[2] and FICO Enterprise Security Scores[3]). The Open Group's Open Trusted Technology Provider Standard (O-TTPS), aka ISO/IEC 20243, is an "open standard containing a set of organizational guidelines, requirements, and recommendations for integrators, providers, and component suppliers to enhance the security of the global supply chain and the integrity of commercial off-the-shelf (COTS) information and communication technology (ICT)." [OpenGroup 2014]. Reddy [2014a] discusses collaborating between industry providers and their customers across the supply chain to address taint and counterfeit items. Reddy [2014b] discusses the use of criticality analysis by commercial-off-the-shelf (COTS) suppliers, combined with a scalable analysis of supplier risk during acquisition, to address supply chain risks. OSS can attempt to achieve a badge through the Linux Foundation CII Best Practices project [LF2017], which establishes criteria primarily on a product's development processes. There are many other articles related to SCRM.

---

[2] https://www.bitsighttech.com

[3] http://www.fico.com/en/products/fico-enterprise-security-score

There are a variety of approaches focused on improving the measurability of security.[4] For example:

- The Common Vulnerabilities and Exposures (CVE) is a list of entries of publicly known cybersecurity vulnerabilities, each containing an identification number, a description, and at least one public reference.[5] CVE is used by many, including the including the U.S. National Vulnerability Database (NVD).
- The Common Vulnerability Scoring System (CVSS) "provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity."[6] CVSS provides a "qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes."
- The Common Weakness Enumeration (CWE) is a list of software weakness types.[7]
- The Common Weakness Scoring System (CWSS) scoring is similar to CVSS but scores weakness types with prioritization instead of focusing on individual vulnerabilities.
- The Common Attack Pattern Enumeration and Classification (CAPEC) catalogues attack patterns. This complements the CWE, since weaknesses in the abstract may not inform system owners sufficiently to understand which software attacks tie to particular weaknesses.
- The Common Weakness Risk Analysis Framework (CWRAF) provides a framework for scoring software weaknesses in a consistent way in the context of various business domains.[8] CWRAF can be used to build a scenario or vignette of the particular system or mission purpose that is most important to the developers and system owners. This allows a measurable prioritization with a resultant focus on attacks that could have the most impact on the systems. CWRAF combines with CWSS to allow this customization of sets of attacks and their technical impacts that are most relevant to the system purpose or mission. A vignette might be crafted that would focus on supply chain attacks that could impact developed source code and allow for the "execution of unauthorized

---

4 For an overview of many approaches, see https://makingsecuritymeasurable.mitre.org/

5 CVEs are explained and listed at https://cve.mitre.org/

6 For more, see the Forum of Incident Response and Security Teams (FIRST) page of the Common Vulnerability Scoring System (CVSS) Special Interest Group (SIG) at https://www.first.org/cvss/

7 See http://cwe.mitre.org/data/

8 See http://cwe.mitre.org/cwraf/

code or commands" that could in turn lead to a "Denial of Service: unreliable execution" or to "Hide Activities."[9] These lower level impact descriptions provide a more detailed refinement compared to the traditional triad of confidentiality, integrity, and availability (CIA).

Organizations developing software often find it valuable to begin by working to counter the "most important" software vulnerabilities. Two widely used lists for this purpose are the Open Web Application Security Project (OWASP) top 10 (for web applications)[10] and the CWE/SANS top 25 list.[11]

Mueller [2012] notes the concern that, "many development tools operate at the same protection level as the operating system kernel and function quite nicely as a [way to deposit] malicious software. It also provides some hints for creating a secure SDE, e.g., noting that inspections must occur only after the source code is placed under configuration control, or the developer could simply add "the malicious functionality after the source code passes inspection or provides the inspection team a listing not containing the malicious functionality." However, [Mueller 2012] also refers in many cases to obsolete technologies or practices (e.g., BASIC's peek/poke, CVS, and programmers "reserving" source modules), so it should be considered in that context.

In [Wheeler&Reddy 2015], we noted that protecting the SDE and countering supply chain attacks is important, and briefly discussed in that material some approaches for countering attacks. Both Wright [2014] and Wright [2017] discuss protecting ("locking down") the SDE. Papers that discuss the security issues of VC systems include [Woiciechowski 2013] and [Wheeler 2015-scm].

A key security principle is "least privilege," which grants programs or users only the privileges they need to accomplish their assigned tasks [Saltzer&Schroeder 1975]. One way to implement least privilege is to implement some kind of "sandbox." Broadly speaking, a sandbox can be defined as a mechanism that restricts a running process to a subset of the privileges and access rights of the invoking user. The concept of "sandboxing" processes is not new, but there has been an increase in interest in sandboxes [Simpson 2011]. Later in this paper, we will discuss using sandboxes to limit tool privileges.

Now that we have reviewed past issues and various policies, we can begin discussing the problem this paper strives to address.

---

[9] Quotes and capitalization are used here because these are references to specific CWRAF entries.

[10] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[11] http://cwe.mitre.org/top25/ and https://www.sans.org/top25-software-errors

3. Problem

We want to develop scalable technical measures to reduce risks from SwA tools, so that projects can use and update a large number of SwA tools with much greater confidence and acceptable risk. To do so effectively, we must understand the problem better.

This section discusses the underlying problem in terms of threats, assets, overall environment, SCRM issues in general, and overall security goals.

Some of the approaches for handling SwA tools may apply to other tools as well, but our focus is on SwA tools. As noted earlier, unlike many other kinds of tools, there is value in having a large collection of different SwA tools, including multiple tools in the same category. Also, SwA tools tend to be updated frequently. In contrast, typically few compilers are used, and compilers are often updated more cautiously. Editors must be trusted, but there are usually few, and they require modification rights to source code (something that SwA tools typically do not require).

A. Threats

CNSSI 4009 defines "threat" as "any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, individuals, other organizations, or the Nation through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service." [CNSSI 4009]

The National Initiative for Cybersecurity Careers and Studies (NICCS) glossary defines "threat agent" as an "individual, group, organization, or government that conducts or has the intent to conduct detrimental activities" [NICCS]. OWASP defines "threat agent" similarly but more broadly, as "a group of attackers that carry out an attack. They can be human (intentional or unintentional) or natural (flood, fire, etc...)." [OWASP 2008].

There are many potential threat agents to an SDE. For purposes of this document, we group threat agents into the following categories:

- **Supplier (of tool)**. These are suppliers of tools used in the SDE, including compilers, editors, VC, and SwA tools.
- **Supplier (of code/data)**. These are suppliers of third-party code other than tools (e.g., to be incorporated into a final product), as well as suppliers of data. By

"data," we mean any information (e.g., geospatial data) provided by a third party that is not third-party code or a tool.

- **Authorized Developer**. These are individuals authorized to use the SDE to develop software. We use "developer" in a broad sense to include requirements engineers, designers, and test engineers.
- **Outsider**. These are individuals who do not meet the previous categories. This includes attackers from outside an organization, but may also include a janitor who works for the company but is not authorized to make changes to the software. One challenge is that attackers outside an organization may work with those inside to make it easy to attack.

B. Assets

Threats are irrelevant unless there are assets or services that need to be protected.

Typical assets that may need some kind of protection in an SDE are:

- **Software code**. This can be subdivided into source code and executables:
  - Source code includes the source being developed and third-party source that is incorporated into the final product.
  - Executables includes executables generated from source code and any third-party executables that are incorporated directly into products.
- **Software tools**. These are the tools used to manage the software code. These can include editors, compilers, and software analysis tools (such as source code vulnerability scanners). These tools might not each require the same privileges. For example, an editor must be able to modify source code, while a SwA analysis tool typically will not.
- **Data**. This is other information managed in the SDE not including software code and software tools. This includes requirements documents, version history, test data that will be processed by software being developed, and test results.

In all cases, these assets may be custom or reused. Reused components, aka off-the-shelf (OTS), can come in many forms, including government-off-the-shelf (GOTS) or commercial off-the-shelf (COTS). COTS may be proprietary or OSS.

The kinds of protection assets needed may differ in terms of confidentiality, integrity, availability, and non-repudiation. The level of protection necessary to protect assets depends on the impact of any failure to protect (e.g., lost money, property, and lives). For example, it may be vital to protect the confidentiality of some custom software, but confidentiality may be irrelevant for OSS (since it is publicly released

already). Backups are often useful to help protect against loss of availability, but may not be adequate to counter loss of integrity.

If code signing is supported, the code signing keys are a particularly sensitive kind of data. The public key infrastructure (PKI) tools supporting code signing are a specialized kind of software tool. There may be separate "development" keys with fewer privileges than final "release" keys. Keys for signing final release software are often kept completely separate from normal development and isolated from any network.

Modern software development depends on third-party software, both as tools and as reusable code that will be embedded in the final products. Third-party software may contain unintentional vulnerabilities. In a worst-case scenario, the third-party software may be malicious because the supplier was intentionally malicious or because someone else subverted the supplier's SDE or supply chain (either upstream or downstream).

C. Overall Environment

There are many different kinds of SDEs. An SDE can be as simple a single laptop managed by a single developer, or it can involve thousands of developers spread across the globe. An SDE may be contained entirely within physically protected environments (possibly connected using strong encryption), or it may extend outside physically protected boundaries. Larger SDEs may be implemented using local SDEs that communicate with each other. For example, SDEs used directly by developers may communicate with many staged specialized build or test environments, which themselves may be centralized or may be managed autonomously. External partners may be considered part of an organization's SDE, whether they are completely integrated or have restricted or occasional access to only specific capabilities.

Today, cloud infrastructures are often used to implement a variety of capabilities, including part or all of an SDE. Clouds have advantages and disadvantages, which must be weighed for the circumstance. Their advantages are apparent; they are often less expensive (due to the sharing of services), may provide higher reliability (due to redundancy), and often provide faster resource allocation (since the resources are immediately available).

However, there are also risks when using clouds, depending on how much is shared. Clouds can be public, community, or private clouds. All clouds are subject to risk from attack, but public and community clouds have a higher sharing risk because of the increased numbers of entities who share the same platform. Those other entities may be subverted or controlled by an attacker. Private clouds significantly reduce the sharing, which can reduce these risks, but private clouds typically provide fewer resources at greater costs.

Clouds typically employ isolation mechanisms such as containers or virtual machines to separate user data and processing. We will discuss the use of these isolation mechanisms, particularly in how to use them to implement tool isolation, in section 5.D.

D. Supply Chain Risk Management (SCRM)

NIST SP 800-161 notes that "Information and Communications Technology (ICT) relies on a complex, globally distributed, and interconnected supply chain ecosystem that is long, has geographically diverse routes, and consists of multiple tiers of outsourcing." It defines ICT SCRM as "the process of identifying, assessing, and mitigating the risks associated with the global and distributed nature of ICT product and service supply chains." [Boyens 2015]

Committee on National Security Systems (CNSS) Directive 505 [CNSS 2017] notes that the "U.S. Government must address the reality of a global marketplace which provides increased opportunities for adversaries to penetrate, and potentially manipulate, information and communications technology (ICT) supply chains. Adversaries seek to subvert the elements or services bound for U.S. Government critical systems to gain unauthorized access to data, alter data, undermine functionality, interrupt communications, or disrupt critical infrastructures."

SCRM is a subset of overall risk management (RM). SCRM, in turn, considers the supply chain risks imposed by the supply chains of products and services. These products include the subcomponents that will be incorporated into the final product being developed. These products also include the tools used during development, such as SwA tools — a point that is often overlooked.

Software suppliers often unintentionally develop products where vulnerabilities remain (e.g., allowing an attacker to seize unauthorized control of the tool or to exfiltrate data from the tool). Suppliers may intentionally insert malicious code, or their development or distribution processes may have been subverted by another party to intentionally insert malicious code. Malicious code can even be designed to attack reused or custom software in another development environment.

Some suppliers, products, or services are riskier than others. Thus, any consideration of them should consider probability and impact and examine trade-offs in terms of cost and benefits, including the cost and benefit of countermeasures. Section 2.B discusses some of the approaches that have been discussed or implemented for addressing SCRM.

E. Security Goals

There are many potential security goals, depending on the circumstance.

First and foremost is the need for the system or capability to survive and operate in its intended cyber environment, which is likely to be contested by adversaries at various levels. Cybersecurity, including SwA, must be built into the system throughout its lifecycle, and maintained and preserved at all phases of its development and operation.

For joint warfighting systems, these cyber survivability requirements are established through the system's System Survivability Key Performance Parameter (SS KPP), in which specific system cyber survivability requirements are described in their applicable capabilities documents. Cyber survivability requires that mission critical functions and information flows be identified, and their components and implementation protected against threats and adversaries commensurate with the impact of their loss or compromise. These components include the system's software. The system's architecture will typically be designed to partition and protect these mission critical functions and information flows into more defensible partitions, with more controlled attack surfaces and interfaces. SwA should be consistent with the mission assurance objectives of the system and its capability including systems-of-systems aspects. For each mission critical function, partition, or component, security goals can then be determined. [Rowell] [Ahner 2017]

Security goals may be organized using the classic confidentiality, integrity, availability (CIA) triad, along with non-repudiation (of senders and receivers) as a separate goal to ease discussion of issues specific to non-repudiation. The CIA triad is often extended this way (e.g., DoDI 8500.01 adds non-repudiation[12] and authentication to the CIA triad [DoDI8500.01]).

Determining which security goals matter depends on many factors, including the threat agents (including their purpose and resources). In practice, these goals need to be segmented further by threat agents and their purposes to ensure that all aspects of these security goals are considered.

In this paper, we summarize potential segmented security goals for an SDE using the matrix in Table 3-1. This matrix shows the overall security goals (CIA and non-repudiation), segmented by different threat agents (complete outsiders, suppliers of tools, suppliers of code or data, and authorized developers) and the threat agent purpose (intentional or unintentional). The supplier threat agents cover risks due to the external supply chain. Note that for simplicity, we omit "unintentional outsider" since if an intentional outsider can be countered, presumably that counters unintentional ones as well. The shaded cells show the intersection among each security goal, threat agent, and threat purpose. The rest of this paper focuses on the "supplier-tool" rows (shown in orange), as opposed to the other areas (shown in yellow).

[12] DoD Instruction 8500.01 spells non-repudiation as "nonrepudiation" (without a dash).

Table 3-1. Security Goals and Threat Agents (General)

Security Goal/ Threat Agent	Threat Purpose	Confidentiality	Integrity	Availability	Non-repudiation
Outsider	*Intentional*				
Supplier-tool	*Unintentional*				
	Intentional				
Supplier-code/data	*Unintentional*				
	Intentional				
Authorized developer	*Unintentional*				
	Intentional				

Within each intersection, we should consider all relevant assets that need to be protected. For example, when a supplier provides a tool, we may need to worry that the tool supplier is intentionally inserting malicious code that could cause the loss of integrity of custom software being developed in the SDE.

4. Weaknesses of Common SDE Technical Approaches

Unfortunately, some common technical approaches to implementing SwA tools in an SDE do not necessarily address the security goals discussed in section 3.E in an adequate way. This is most easily shown by example. In this chapter, we briefly describe several common approaches, and then discuss their weaknesses. Of course, whether or not an approach is acceptable depends on the risk. In some cases, these weaknesses are acceptable, but in others they are not. We will discuss in Chapter 5 what can be done if the risks are unacceptable.

A. Example 1: Analysis Remotely Executed

One approach to implementing SwA tools in an SDE is to have the SwA tool run remotely in an external environment, as illustrated in Figure 4-1. Note that this requires that the SDE send whatever is to be analyzed to an external environment (e.g., source code (including build instructions), bytecode, and executables). Here, we assume that the communication path between the external environment and SDE is protected (e.g., by HTTPS using transport layer security (TLS) with reasonably secure settings).

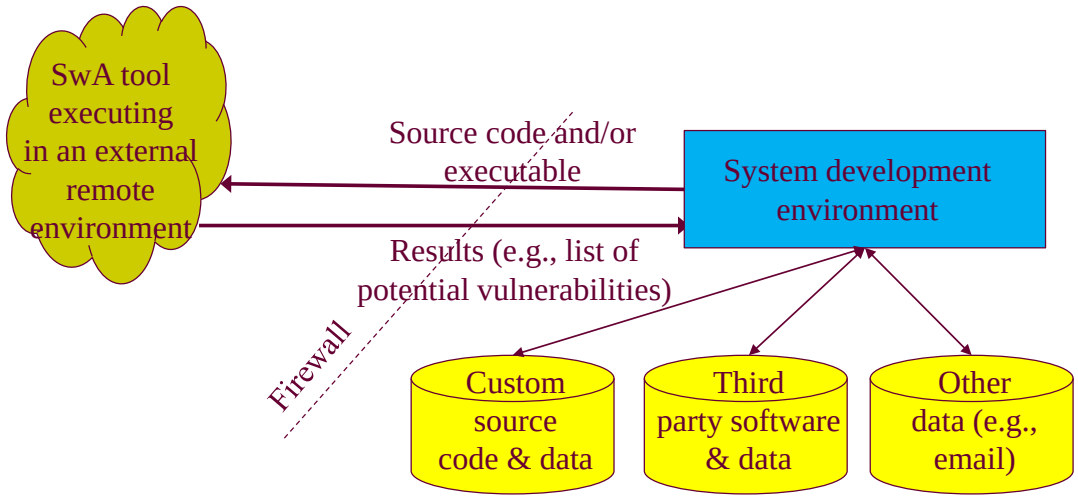


Figure 4-1. Analysis Remotely Executed

Many SwA tool suppliers prefer this approach. It doesn't reveal their own source code or methods, they do not need to develop software installation systems, and it is easy

to reliably charge tool users for use. Updating the SwA tool in this approach is extremely easy, relatively low cost, and immediate.

Many commercial organizations using SwA tools also like this approach, because the organizations can avoid the delays and costs of installing a tool. They can instead immediately use the SwA tool.

However, in many circumstances there are serious risks to this approach:

- This approach provides additional avenues for loss of confidentiality in both the material sent to the SwA tool (such as source code), as well as the results of the tool. This approach trusts the SwA tool supplier and suppliers of any computing resources. If they are untrustworthy, or subverted by attackers, that material can be sent to adversaries. This approach also risks the loss of anonymity of the developing organization. This is fine if confidentiality is not needed (e.g., it's OSS in development), but in other cases this is unacceptable. In particular, this is often unacceptable for classified and controlled[13] code unless the suppliers of the tool and computing resources are approved for this purpose.
- There is a risk in integrity of the results being sent back to the SDE. The SwA tool supplier, if untrustworthy, might intentionally omit results (such as a vulnerability report) and instead provide that to others.
- In some cases this may raise serious legal questions. Many systems include third party software, and their licenses may forbid sending that software to external parties without further contract adjustments.

This approach is fine for some use cases but inappropriate for others.

B. Example 2: Locally Executed with Internet Access

Another approach to implementing SwA tools in an SDE is to install and run the SwA tool within a local SDE, as illustrated in Figure 4-2. For this example, we presume that the SDE is connected to the Internet (possibly with firewalls and monitoring systems) and that the SwA tool has unrestricted access to the source code and executable.

[13] For our purposes, controlled code is code that has restricted distribution (e.g., is labelled For Official Use Only (FOUO), is restricted under International Traffic in Arms Regulations (ITAR), or has distribution markings).

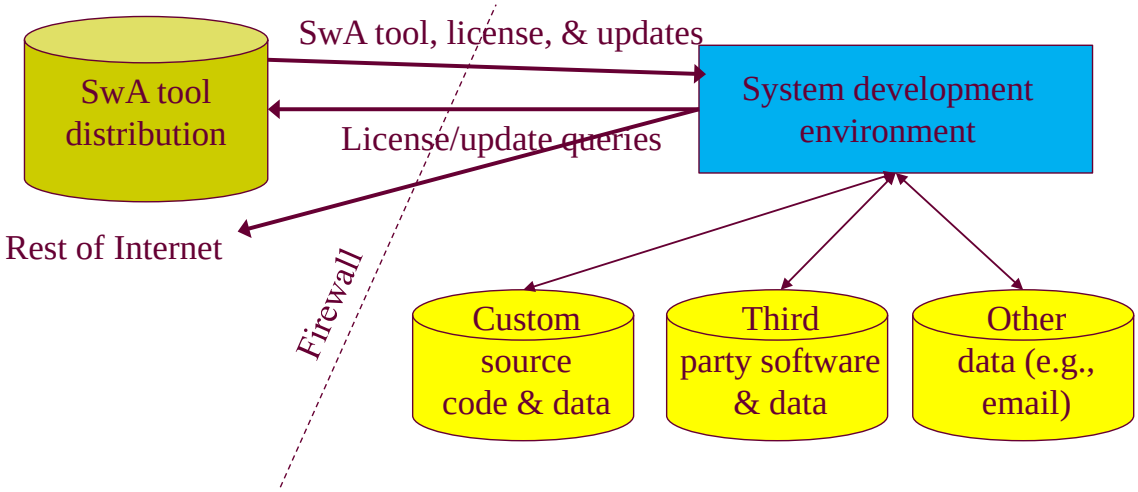


Figure 4-2. Locally Executed

One advantage to this approach is that the material sent to the SwA tool (such as source code), as well as the results of the tool, are not expected to be sent outside the organization's SDE. Some anonymity can often be achieved where desired, by having another organization buy and/or download the SwA tool on the developing organization's behalf.

However, there are risks to confidentiality. An untrustworthy SwA tool could still send outside the SDE various information such as materials for analysis (e.g., source code) or results. The SwA tool might have additional functionality (such as "report on crash") that could cause release of this data. The SwA tool might even send this information maliciously. The information sent might not be the source code itself; it could instead be a security defect that was detected by the tool but not reported to its user. This exfiltrated data can also damage anonymity (where applicable), even if the tool is bought through another party. In many cases such exfiltration could be easy to achieve, since it could be hidden in other actions such as during tool or ruleset updates.

There are also risks to integrity. Unless there are other safeguards in place, these tools often run with the privileges of the developer, which means that the SwA tool may be able to modify information, such as the code being developed or managed. The SwA tool might also be able to modify other tools. Any of these modifications could result in the insertion of errors or malicious functionality.

This approach is fine for some use cases but inappropriate for others.

C. Example 3: Individual Roaming SDE

In many commercial situations, individual software developers have their own computers (typically laptops) that can roam. These computers routinely connect and

disconnect from the Internet. This flexibility enables developers to routinely work anywhere they want, including home, coffee shops, and while traveling. This might be the primary way for using the SDE or merely an extension of the environments described in the previous examples.

It would be possible to require that the SDE be in a separate trusted environment, contacted via the Internet, and that these roaming computers would not have local SDEs. This may be called "remote access." The developers' computers are then windows into the SDE, rather than having an SDE itself fully local. The trusted environment might be within an organization, and/or implemented by a cloud provider, but in either case, the SDE is safeguarded by their physical and logical protections. The developer's computer may be stolen or subverted, but the primary SDE is somewhat protected. However, remote access requires that the developer have a continuous Internet connection to get work done. In many circumstances, continuous Internet connectivity is unavailable or problematic.

Some organizations instead allow developers to have an SDE within their own roaming computer, and that development can occur even without continuous Internet connectivity. Supporting work while disconnected from the Internet may be called "offline access." To support this, the developer's computer must have a working SDE, complete with local source code copies as needed for modification, as well as many tools (such as editors). This has become a major and common trend in the software industry, supported by technologies such as distributed VC systems (such as git).

Offline access has many advantages, but it also has additional risks. Far more information is on the local system, so far more information can be lost if the developer's computer is stolen. Unless special measures are taken, there are typically no limitations on local tools, including the SwA tools. As a result, all the risks of Example 2 often apply, and the developer's local system is often less well protected.

A very common risk countermeasure is to use HTTPS for a developer to connect back to a remote site. However, by itself this does not provide traffic monitoring of a developer's computer. HTTPS can also be used to unwittingly connect to sites that include malicious attacks (the attacks may be intentional or indirect via systems such as ad networks).

Another common risk countermeasure is to use a virtual private network (VPN) from the developer's computer to a remote system. This is often paired with the use of an HTTPS proxy, so that HTTPS traffic can be intercepted, decrypted, and re-encrypted by the organization. When used, this enables traffic monitoring for dangerous activities and makes it easier to prevent access to malicious sites. However, this only works when the VPN is active. VPNs and HTTPS proxies insert network traffic delay. In addition, a VPN

cannot prevent attacks due to a SwA tool itself, since the SwA tool is typically already running within the SDE.

D. Example 4: Physically Secured Development Environment within Disconnected Private Network

A completely different approach to securing an SDE is to physically secure the SDE within an organization's facility and use a private network that is not connected to any external network, including the Internet. In many cases, there is no external network connection at all. The physical security may be implemented with locked buildings, separate physical areas for development teams, armed guards, fences, TEMPEST shielding, and so on. This approach is often used for classified work and/or where SwA tool approvals are slow (because they cannot connect to a larger network without the approvals).

Clearly this reduces many risks, since it is more difficult for an adversary to penetrate this SDE. On the other hand, this may also gravely reduce productivity. In addition, the SDE must still bring in external data (e.g., software updates and new libraries for use). These mechanisms for transferring data are generally called cross-domain solutions (CDS) and may be implemented in a variety of ways (e.g., using a high-assurance guard). However, CDS rely on people and technology to determine if the transfer is acceptable, and this is by no means guaranteed to be perfect. Indeed, the software brought into the isolated SDE could be malicious. These occasional data transfers are necessary yet reduce the effectiveness of this approach.

In some circumstances, it would be wise to consider other solutions. The next chapter will discuss how to develop alternative solutions.

5. Developing Secure SDE Solutions

The SDE, including the SwA tools, should be designed and implemented to meet its requirements, including its security requirements. The challenge is that some organizations focus on the software to be developed and fail to consider the security requirements of the SDE itself.

The best approach for addressing SDE security requirements depends on the risk, and thus depends on the threats, the assets to be protected, and the potential impact. It also depends on the distribution of developers (e.g., it may vary based on whether or not there are remote developers). We will discuss sample solutions later, but we will first describe how to develop solutions.

As noted in section 3.A, SwA tool suppliers are potential threat agents, and this may be a surprise to some. SwA tools can have unintentional defects, and some of them come from less trustworthy sources. What's more, SwA tool suppliers may themselves be attacked and subverted.

Supplier trustworthiness is not necessarily a problem, because it's quite possible to design the overall SDE to reduce the trust that must be given to SwA tools. Approaches such as isolated environments, limited privileges, and limiting data transfers can reduce risk in a cost-effective way. These approaches can also be used to address risks from other kinds of tools within the SDE. That said, other tools (such as editors and compilers) have very different characteristics from typical SwA tools; in this paper, we focus on SwA tools.

A. Security Principles

Saltzer and Schroeder identified key security design principles in their seminal work [Saltzer&Schroeder 1975]. These principles also apply to the SDE in general, and some are particularly relevant to the solutions discussed in Chapter 5:

a. Economy of mechanism: "Keep the design as simple and small as possible."

b. Fail-safe defaults: "Base access decisions on permission rather than exclusion."

c. Complete mediation: "Every access to every object must be checked for authority."

d. Open design: "The design should not be secret."

e. Separation of privilege: “Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.” Today, we would call this “two-factor authentication.”

f. Least privilege: “Every program and every user of the system should operate using the least set of privileges necessary to complete the job.” This limits damage. In section 6.A.2, we discuss creating computational environments with limited privileges for SwA tools, to limit the damage that they can cause.

g. Least common mechanism: “Minimize the amount of mechanism common to more than one user and depended on by all users.” In section 6.A.2, we discuss creating computational environments with *copies* of data, so that originals cannot be compromised.

h. Psychological acceptability: “It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.” We will later discuss automation to ease developer use and ensure consistency.

An additional secure design principle is the use of input validation, even though it is not expressly identified in [Saltzer&Schroeder 1975]. Input validation is the practice of checking all input data for appropriate formatting and ranges of values. We will later discuss the potential role of validating data from SwA tools before they are accepted into other circumstances.

All of these secure design principles are relevant. In the context of protecting against unintentional and intentional vulnerabilities in SwA tools, the principles of least privilege, least common mechanism, and input validation are especially important.

B. Project Policies

Projects will need to select various policies depending on the security goals and threats, as well as trade-offs in cost, schedule, and performance.

For example, if the software being developed has very strong confidentiality requirements, management may choose to have a project policy that the software being developed must not ever leave a controlled physical environment. DoD requires this when developing classified software. This policy can be costly, slow development, and even prevent many developers from participating, so the trade-offs of this policy decision should be carefully considered when a trade-off is allowed. A policy to restrict development locations could be considered to be grounded in the fail-safe defaults principle (because allowing external development expands the attack surface) and the

least privilege principle (because allowing external development may enable attackers to contact the attack surface unnecessarily).

Another example might be to have releases always be signed using a private key that is never placed on a computer connected to the Internet. Such a policy would be grounded in the least privilege principle.

C. Designing the SDE for Security

The project's SDE needs to be designed to implement the project's security goals against the threats the project's SDE needs to counter. Completely addressing the security of the entire SDE is large and beyond the scope of this paper, but we cannot ignore the larger SDE issues since they may create bigger problems. Thus, designing the SDE for security is foundational.

In general, the SDE should be designed to meet the security goals against various threat agents, as shown in Table 3-1. This section covers some aspects of designing the SDE for security, including some aspects that may not be obvious. To secure the SDE, the designers and maintainers of the SDE must determine which combinations of security goals, threat agents, and threat purpose are relevant and address each relevant combination with adequate countermeasures. What is adequate for one situation may inadequate for another, so the strength of the countermeasures must be considered.

Although this paper emphasizes SwA tools, here are some approaches or issues you could consider when trying to cover the security goal, threat agent, and threat purpose in designing and maintaining an SDE for security:

- Isolate the development, build and test environments and possibly other environments to limit damage (see section 5.D):
  - Use many test environments, all isolated from the final build environment.
  - Ensure that a malicious SwA tool can't exfiltrate or modify source code, tools, etc.
  - Isolation can be implemented by separate machines, virtual machines, and/or containers, possibly implemented on a private or hybrid cloud. In some cases, this can speed building and testing by providing more computing resources.
  - SwA tool results could be provided using isolation mechanisms such as viewing a virtual machine display, HTML display, etc.
- Harden software against attack to the maximum extent practical. This includes the virtual machine monitors, operating systems, runtime libraries, container systems, etc. You can harden software using security configuration guides such as Security Technical Implementation Guides (STIGs). This is more effective if combined

with configuration scanners provided with proper permissions to deeply analyze the system.

- Control communication mechanisms within the SDE environments (see section 5.E). For example, use diodes for one-way information transfer, which ensure that data only transfers in one direction. These may be procedural or electronic.
- Cache third-party information (including tools and other software) locally for trusted access:
  - Provide controlled/protected transfers.
  - Check digital signatures of third-party information before any use.
  - Maintain availability when Internet access denied.
  - Enable review.
  - Alert and possibly prevent use of known problematic versions.
- Ensure configuration management is maintained, including version control (VC) that is hardened against attack:
  - Limit read/write access to developers authorized to do so.
  - Record and digitally sign every commit, including who, what, and when. This enables tracking the actions ("attack attribution") of a malicious developer or a subverted developer's account. This disincentivizes attacks by a malicious developer and eases later recovery should a malicious developer succeed.
  - For more information about security and VC systems, see [Woiciechowski 2013] and [Wheeler2015-scm].
- Consider using SwA tools to examine and counter other tools. SwA tools can sometimes to be used to examine each other and/or their proposed changes. In many cases this is not practical (e.g., due to cost).
- Perform backups to enable restoration.
- Protect communication links in and out of the SDE:
  - Use encrypted links.
  - Use redundant links.
  - Use an anonymizer. This limits the revelation to outsiders of who is doing what.
  - Use organizational proxies and blind buys to prevent suppliers from knowing exactly what a product will be used for; for example, when purchasing something, purchase as "US government" or as a large contractor, not as a

specific project (which might be targeted). This is impractical in many cases; for example, sometimes it is important to work with a supplier so that the supplier can provide the right products and information for proper use.

  - Use a firewall.

- Monitor the SDE for abnormal and malicious behavior:
  - Detect unusual/unexpected behavior (e.g., by malicious developer).
  - Counter what gets through, because no prevention is perfect.
- Develop a recovery plan if SDE is compromised.
- Sign releases of software:
  - Review and certify protection of release signing infrastructure.
  - Ensure that developer signatures are not the same as release signatures.
  - Strongly protect release signature keys (e.g., ensure their private keys are offline).

These often depend on isolation mechanisms and communication mechanisms, so we will briefly discuss each in turn.

D. Isolation Mechanisms

Implementing many of these security design principles requires some kind of isolation of the less-trusted functions (e.g., a SwA tool) from other parts of the SDE. For example, least privilege requires that the less-trusted functions do not have concurrent access to both the Internet and private source code unless that is needed. These isolation mechanisms for computing environments (CEs) are key for running code while limiting privilege.

There are many different kinds of different isolation mechanisms. These include:

- Isolated computer ("air-gapped" computer): a computer used in isolation with no network connectivity and possibly with other physical protections (e.g., TEMPEST shielding against electromagnetic radiation, isolated power, and so on). An isolated computer can still be attacked through input/output systems (e.g., USB connections when they occur) or through side channel attacks (e.g., via noise).
- Bare metal server: a physical server dedicated to a single tenant.[14]

[14] https://www.rackspace.com/en-gb/library/what-is-a-bare-metal-server

- Virtual machine (VM): an efficient, isolated duplicate of [a] real machine [Popek 1974]. VMs are implemented by a hypervisor, aka a virtual machine monitor (VMM). Hypervisors may be implemented directly on hardware (aka "native," "bare metal," or "type 1") or as an application (aka "hosted" or "type 2").
- Container: A process running on a shared operating system kernel that is isolated from other processes. A container runs a container image, which is a "lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings."[15] Containers are also called "zones" or "jails." The ability to run containers is also called operating-system-level virtualization.
- Chrooted process: A process running on a shared operating system kernel that has its filesystem isolated from other processes using the chroot system call. The chroot system call was introduced to Unix version 7 in 1979 [Bell Labs 1979], and is widely available in Unix-like systems (including Linux). Some consider a chrooted process to be a kind of container (aka operating-system-level virtualization), while others consider a container as something distinct from a chrooted process.
- Separate user account: Sets of processes are isolated from each other by assigning them to different user accounts. Most operating systems have built-in mechanisms for isolating users and protecting them from each other. For example, on the Android operating system, different applications are assigned to different user accounts to separate the privileges of applications.

Different isolation mechanisms tend to be stronger or weaker at isolation, and different implementations will differ in the quality of their isolation. The list above is in approximate order of strength, from strongest to weakest. For example, virtual machines tend to provide stronger isolation than container-based systems, because virtual machines do not share a single underlying operating system kernel, but virtual machines also require more resources. In contrast, container-based systems are much faster to start and tend to require fewer resources, but because they share a single underlying operating system kernel with far more interfaces, they tend to provide weaker isolation [Wheeler 2017cloud]. That said, these orders are only approximate; a well-hardened system of any kind is typically far more resistant to attack than a poorly configured one.

The right isolation mechanism for a particular situation depends on the risk and cost. Attackers may develop software to work around or subvert an isolation mechanism. Properly configured strong isolation mechanisms, especially those in series (requiring multiple vulnerabilities to get through), can provide a strong defense against attacks.

---

[15] https://www.docker.com/what-container

As discussed later in Chapter 5, we can use these isolation mechanisms to reduce the privileges granted to SwA tools. For example, we can put a SwA tool in its own virtual machine or container, limiting the access it has to other resources (including the Internet).

The BSIMM activity "Use application containers" (SE3.4) notes the value of using application containers to support security goals, but the BSIMM survey version 8 also finds that only 4% (4/109) of the surveyed firms do this [BSIMM 2017]. Application containers are used in organizations today; drivers for their use include "ease of deployment, a tighter coupling of applications with their dependencies, and isolation without the overhead of deploying a full OS on a virtual machine." However, BSIMM only counts application containers if they are used to support the organization's software security goals, and for BSIMM's purposes "containers used in development or test environments without reference to security do not count." That said, BSIMM notes that, "containers provide a convenient place for security controls to be applied and updated consistently," so we expect that their use will increase over time.

E. Communication Mechanisms

Totally isolated environments are rarely useful. Instead, we typically want environments that are *mostly* isolated but permit specifically approved communication mechanisms. In general, the approach to communication depends on risk and cost.

Here are some options for communicating between environments:

1. View one environment's results and manually respond in another. This could use, for example, a secure Virtual Network Computing (VNC) display or a virtual machine display. This strongly isolates the environment, but does require extra work for a developer to separately jump to the "same place" in the software editing environment.
2. View an environment's outputs by bringing them into a protected viewer. For example, the isolated environment could run a webserver viewable only by the main SDE; the SDE's web browser is used to view the data and is trusted to provide a defense against whatever the isolated environment does.
3. Copy/paste text between environments (where supported).
4. Create "diffs" (proposed changes) in sender, transfer them (e.g., as a file), and review them in the receiver before acceptance. VC systems can commit these changes, perhaps marked in some way so they can be reviewed further or reverted.
5. Allow direct transfer of data from one environment to another, but filter it (this enables automation and can restrict damage but provides less isolation). This

filter must use whitelist to check the data and help counter attack. Filtered output may need to be processed by a trusted tool in the receiving environment.

6. Allow direct transfer, but log/monitor it (this provides even less isolation).
7. Allow direct transfer without any control (this provides little isolation).

It would be possible for environments to share direct access to files (e.g., of the source code to be analyzed). However, this sharing is a violation of the "least common mechanism" principle. It is typically safer to send copies of source code to the SwA tool, so that it cannot directly modify the "real" code.

F. Solution Implementation

Once a solution has been designed, it needs to be implemented. Here are a few notes about implementing a secured SDE — in particular, for securing SwA tools.

Ensure that your isolation mechanisms are ready before putting the SwA tools in them. Test the SwA tools in a safer and/or isolated environment with sample code before using a copy of the real thing. Isolation mechanisms make testing easier, because they limit damage and make it easier to restore to a known state.

As software development teams get larger, there tends to be more specialization. For example, some people may focus on supporting the development environment (or at least the configuration management system). In larger organizations, there might be a specialized team that is in charge of the SDE, including assessing the tool supplier as part of due diligence, assessing tool capabilities, bringing new tools (including SwA tools) into the SDE, and continuously updating tools.

G. Maximize Automation

It is important that tool actions be maximally automated (e.g., with a script), to implement the principles of section 5.A. Otherwise, the tools may not be used correctly.

For example, if SwA tools are started within a separate isolated environment, and then a copy of the source code is sent to that environment, a simple script should do this automatically so that it will be done correctly every time. This automation should be reviewed by others before use, to ensure that it works correctly. In many cases, the automation should be designed to work even if the developer is working at a remote location (where policy permits this). Developers should not need to know the details about how the automation works, and the automation should be reliable so that the developers can focus on their work instead of debugging their tools.

6. Sample Solutions

The best approach to solving the problems noted earlier depends in part on the risk, including threats, assets, and impacts. The overall goal is to manage risk — not necessarily to eliminate risk.

This chapter discusses some sample solutions, building on the weaknesses noted in Chapter 4 and the approach for developing solutions described in Chapter 5. We use the term "sample" because there are many possible solutions, and thus it is not possible to be comprehensive. However, it is possible to illustrate useful solutions to build on, and this chapter provides those samples.

We present two sample solutions, "medium protection" and "high protection." We do not show a "low protection"; if the risk is considered low, typically developers would not work hard to counter the risk. The medium protection approach provides some defenses against malfunctioning SwA tools while not requiring significant resources. The high protection approach provides more (and more costly) defenses and shows how protections against malfunctioning SwA tools can be addressed in a larger context. These are merely samples; you may find that selecting only some elements or using a different approach would work better in your environment.

There are costs to any change, but these changes also have advantages. Most obviously, they counter certain kinds of attacks. Since these attacks are countered, they can have the effect of enabling users to use many more SwA tools with confidence and to update them more regularly. Since using many SwA tools increases the ability to detect problems early, and newer versions of SwA tools are typically better than previous versions, these changes can have the overall effect of increasing the assurance of the software being developed or evaluated. In addition, these mechanisms can be used for tools other than SwA tools, providing a ready mechanism for handling tools that are not completely trusted.

A. Solution 1: Medium Protection

Solution 1 provides medium protection against SwA tools while not requiring significant resources. It does not try to protect against malicious developers.

Figure 6-1 illustrates this medium protection solution. The SDE includes two different computing environments (CEs) for each SwA tool, install/update and analysis. The install/update CE can access the supplier's distribution system, but not the rest of the

trusted software development environment. The install/update CE creates an image with the SwA tool. The analysis CE uses this image, receives the data to be analyzed (such as source code), and produces results.

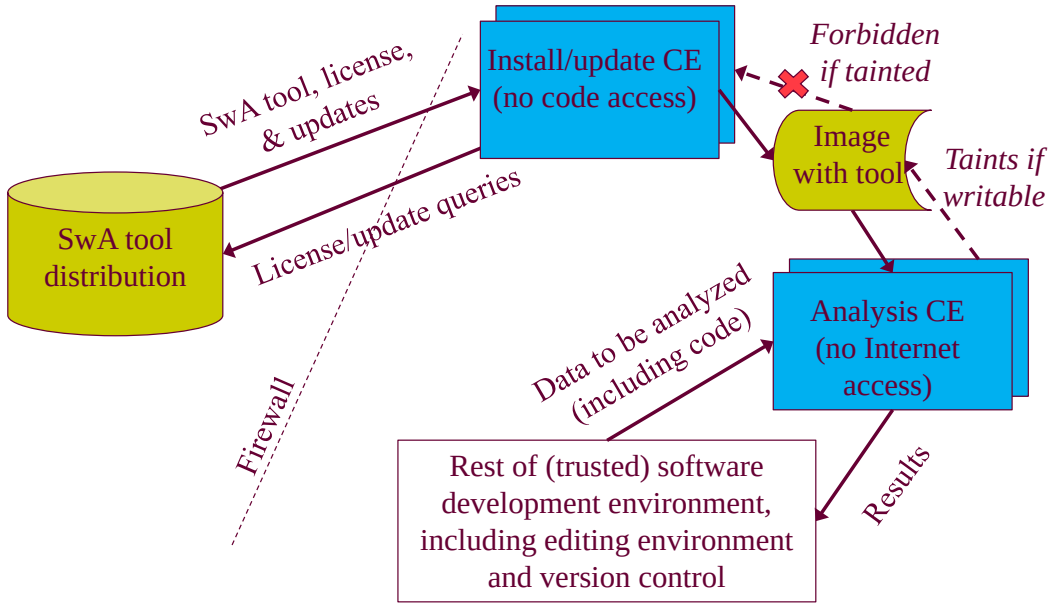


Figure 6-1. Medium Protection Illustration

The following subsections explain this approach in more detail.

1. Properties

Solution 1 has the following properties:

1. Each SwA tool is run within a CE that is isolated from the rest of the development environment. This isolation provides protection from potentially malicious SwA tools to the rest of the (trusted) software development environment. The CE may be implemented as a container, virtual machine, or similar mechanism. In general, containers have faster start-up time and require less storage (they do not duplicate the operating system kernel and can more easily use optimizations like copy-on-write), however, containers typically provide less isolation because there is a shared kernel.
2. Although the CE is isolated in general, it has specifically approved input/output interfaces. In particular, it must have interfaces so that software to be analyzed can be supplied to the CE and results can be retrieved. These interfaces may be implemented as using various mechanisms such as shared directories, shared drives, secure shell (SSH) file transfer protocol (SFTP), and virtual displays.
3. The isolated CE is run within the organization (not within an external public cloud service). Note that in some circumstances, this property might be relaxed.

4. There are two different kinds of CEs to be used with SwA tools, “install/update” and “analysis,” used as described below.
5. The install/update CE creates or updates an image that includes the SwA tool and its data, whereas the analysis CE uses an image. An image may be untainted or tainted; an image is tainted if it could have data from the development environment that should not be publicly revealed. Note that that “taint” in this context has nothing to do with whether or not the SwA tool is malicious.

These properties are used to prevent software source code from unauthorized exfiltration by a SwA tool, unintentionally or intentionally, as well as to prevent a SwA tool from performing unauthorized changes.

2. Designing CEs for Different Purposes

These two kinds of related CEs must be used as follows:

- *Install/update*: When using an install/update CE, the CE must have access to SwA tool installation/update information; in many cases it may be granted access to the Internet to obtain this information. For example, in some cases, the CE may need to create a dynamic query to receive an updated tool, tool data, or tool license.

  Use of this CE produces a new or updated image that contains a new or updated SwA tool and tool data. This image might be transferred to other computer systems (e.g., if analysis is done in a network-isolated development environment). Encryption (e.g., https) and/or digital signature verification is used to ensure that tool and tool updates are those provided by the supplier.

  A key aspect of the install/update CE is that it never has access to any data to be analyzed, such as source code to be analyzed. As a result, this CE cannot reveal confidential code, since it never has access to it. It also cannot perform unauthorized modifications of the code, since again, it never has access to the code to violate its integrity.

  A tool supplier could provide an image that has the tool already installed within it to be run by the analysis CE. For example, the tool may be provided as an Open Container Initiative (OCI) image to be run by a container runtime (such as Docker), or an Open Virtualization Format (OVF) to be run in a virtual machine (such as VMWare or VirtualBox). In this case, there’s no need to run the install/update CE to install the tool, and updates can be performed by directly acquiring an updated image. Accepting this image does not mean that it is totally trusted, for example, be sure that the image will not require or have access to the Internet when run as an analysis CE (as described next).

- *Analysis*: When using an analysis CE, the CE has access to an image containing a SwA tool (e.g., as created through the install/update CE) and data to be analyzed (including source code). However, the analysis CE cannot access the Internet, so it cannot send its data directly to the Internet or use other mechanisms that might reveal that data (protecting confidentiality).

  Images may be *tainted* or *untainted*. When an image with a SwA tool is first created it is considered *untainted*. If an analysis CE has read-only access to an image, then that image does not become tainted. However, if an analysis CE has ever had access to data to be analyzed, and then can gain write access to an image at any point after that, then that image becomes tainted and cannot be used with an install/update CE. This prevents any exfiltration because, although the SwA tool could incorporate analyzed source code in the SwA tool itself (e.g., through caches), that data could not be exfiltrated during the next install/update process.

There are some subtleties involving the image:

1. We have intentionally defined an image as being "tainted" if it could have data from the development environment that should not be publicly revealed. It is typically difficult to determine if writes to an image include unauthorized information, but it is easy to determine if an image is writeable. Thus, we focus on writeability. If an image could have unauthorized data on it, it is tainted. An organization could use a manual review process to review the changes and determine that the image is untainted, though that could be expensive (especially if steganography is to be countered).
2. An image could be copied. For example, an untainted image could be copied to a second image, and only the second image could be made writeable by the analysis CE. The second image would become tainted, but the original image would be unchanged and thus untainted. If the image contains a SwA tool, there may be licensing issues in making a copy depending on the license and whether or not the original image is considered a backup.
3. Some systems make it possible to make an image immutable (read-only) and create a separate "difference image" when writes occur. For example, VirtualBox can do this. If an analysis CE uses these mechanisms, the original image is unchanged (and thus untainted), while the difference image becomes tainted. This approach may be especially useful with some proprietary tools, because in this approach no additional copy of the SwA tool is made (reducing the risk of an unintentional license violation).
4. In some cases an analysis CE may use an image purely as a read-only image, and either provide results back or store them elsewhere (which then become tainted, because they're based on data that should not be released).

3. Handling Results

Naturally, the SwA tool results need to be handled safely. Results from SwA tools vary depending on the tool. Many SwA tools provide reports about vulnerable lines of code or specific inputs that cause crashes. Some SwA tools can provide proposed modifications to the source code (aka changes, patches, or diffs).

Here are a few approaches, depending on how the tool works and the level of acceptable risk:

- Users view the results by only viewing the virtual screen of the analysis CE, for example, using the viewer of a virtual machine's screen or a secured Virtual Network Computing (VNC) client. The underlying interface is limited (e.g., screen image updates, mouse motion within a certain region, and keyboard actions when selected), which limits the attacks that can be performed.
- Users view the results by using their web browser to connect to a web server running on the analysis CE. This tends to have a larger risk, because it opens a larger attack surface. For example, if the web server serves malicious or erroneous pages, the web browser is being relied on to defend against these pages, and web browsers have bigger attack surfaces (e.g., they must be able to securely run JavaScript sent to them by the server).
- Users could download the results and check them before any use. These checks should be performed using a rigorous whitelist filter, for example, validation of XML against a strict XML schema, checking that the encoding is valid (e.g., UTF-8), and checking that only valid characters are allowed for each field. This is riskier, since the filter may allow some attacks through, but it does mitigate others. This presumes that the downloading system will not just execute malicious code during download. The check needs to run on a more trusted system, because we do not fully trust the analysis CE.
- Proposed modifications can be downloaded to the editing environment and manually reviewed before accepting them in the real source code.
- Users could download the results into the editing environment directly. This presents more risks, because the data is being directly imported, though it may be acceptable depending on the type of result and level of perceived risk.

4. Discussion

Some organizations use SwA correlation tools that combine the results of many different SwA tools. In this case, a set of CEs is used, one for each traditional SwA tool and another for the correlation tool. The results of one CE would then be input to another

CE. Thus, the whole approach is scalable, both for using many tools and for combining tools results using yet more tools.

Using different kinds of CEs is somewhat similar to the Biba integrity model [Biba 1975]. In short, using a CE to analyze software "contaminates" the CE from future tool updates. Of course, tools might not maliciously exfiltrate software; it can happen unintentionally (e.g., via caches). This approach greatly increases confidence in which we can use software, without the expense and delay of reviewing the tool's code in detail.

This approach is in some ways similar to Example 1 (section 4.A), in the sense that the tool is run in an environment separate from the rest of the development environment. However, unlike that example, the tool supplier never has an opportunity to see the source code being analyzed. Instead of sending the source code to the tool supplier, the tool is run a separate environment controlled by the developer, and source code is never present when the tool is able to contact the tool supplier (e.g., for updates).

This approach is in some ways also similar to Example 2 (section 4.B), in that the tool is not run within the tool supplier's environment. However, this approach creates a separate isolated environment, instead of being part of a single overall development environment that has no internal isolation mechanisms.

One risk in this approach is that if developers accidentally misconfigure the CEs there could be a loss of confidentiality or integrity. An obvious solution is to maximally automate these steps; see section 5.G.

5. Meeting Security Goals for Solution 1

This approach is focused on countering SwA tool suppliers as threat agents. This is not guaranteed (e.g., it depends on the strength of the isolation mechanisms, the strength of the filtering of results, and developer practices), but it nevertheless reduces risks. The following table shows the security goals, the threat agents against them, and how this approach addresses them.

Table 6-1. Security Goal Coverage by Solution 1

Security Goal/ Threat Agent	Threat Purpose	Confidentiality	Integrity	Availability	Non-repudiation
Outsider	*Intentional*	-	-	-	-
Supplier tool	*Unintentional & Intentional*	Only install/update CE has access to supplier; it has no source code access and it is isolated from analysis CE which has source code access	Analysis CE only has a copy of source code, and is partly isolated from rest of development environment. Results are handled in a risk-based manner, countering attack via tool	Analysis CE can run regardless of state of supplier.	HTTPS/digital signature verification is used to verify that the tool is signed by the supplier's key (the sender). This does not directly supply non-repudiation of the receiver.
Supplier code/data	*Unintentional*	In some cases, SwA tools can help detect/counter unintentional vulnerabilities, but this depends on the tool and data available (e.g., source code analysis requires source code).			-
	Intentional	Addressing this requires the use of SwA tools that can detect intentional malicious code. Detecting previously unknown malicious code is difficult. Mitigating this risk would be typically supplemented by other means.			-
Authorized developer	*Unintentional*	SwA tools help detect/counter unintentional vulnerabilities in custom software. There is a risk of accidentally misconfiguring CE use, which can be countered by maximizing automation.			-
	Intentional	-	-	-	-

Note that solution 1 primarily focuses on countermeasures for SwA tools, as noted in Table 3-1.

This approach does not address all possible security issues. This approach does not by itself counter malicious threat agents who are outsiders, though if the development environment and the SwA CEs are protected (e.g., by firewalls), then outsiders can be countered to a limited extent. This approach has a limited capability to address non-repudiation of suppliers of code/data, since the code/data is not necessarily isolated when it executes in other contexts. Perhaps more importantly, this approach does not counter attacks from authorized developers since they are trusted. Whether or not this matters depends on the security goals.

This approach could be entirely executed within a protected enclave. It could also be implemented on individual developer systems (e.g., on laptops), where the CEs run on

the developer system, and the updates to code could then be sent back to repositories accessible to others (e.g., via encrypted links).

B. Solution 2: High Protection

Here, we present a specific example of a solution that provides higher protection against SwA tools and malicious developers, accepting that more resources are required. This shows how mechanisms to counter malicious SwA tools can be incorporated in a larger solution for protecting an SDE against malicious developers and other problems.

1. Properties of Solution 2

Solution 2 builds on solution 1 (incorporating all of it) but adds additional mechanisms in cases where the costs are justified.

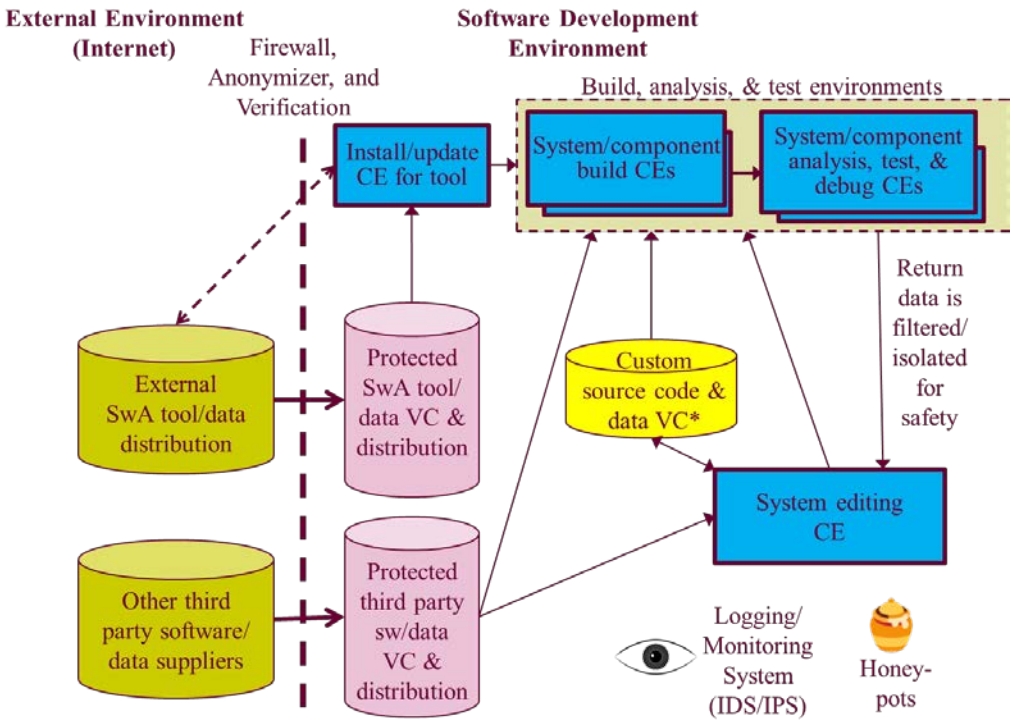


Figure 6-2. High Protection Illustration

Figure 6-2 illustrates this high protection solution, which is more complex than the medium protection solution discussed in section 6.A. This example solution adds the following:

- All third party information (shown in the lower left in Figure 6-2) is retrieved using encrypted links through a firewall. It is retrieved using anonymizers so that external parties, and in some cases suppliers, will not know who is making

the request and thus are less likely to perform an attack. In addition, we suggest that the encrypted links should be redundant with failover (to provide availability).

- All externally retrieved information (tools, code, and data) are verified to be from their expected source before they are used. These integrity checks do not counter malicious suppliers, but they may thwart man-in-the-middle attacks. Here are some verification mechanisms that should be considered (using more than one can make the process stronger):
    - Verify digital signatures before using the information. This requires that the supplier signs the information, and that the receiver has the information to correctly verify the signature. A digital signature is only to be trusted if there is some assurance that the information was created and maintained using key principles like least privilege and need-to-know with keys that are protected. The supplier's developers should not able to easily create the final release builds of production code. The entire digital signing infrastructure of the supplier should be audited for security technology and practices by a team with code signing and development experience.
    - Verify cryptographic hashes of the information to ensure they have correct values. This requires knowing the correct values. These cryptographic hash values may be retrieved from a trusted website that uses HTTPS (but this does not help if the website is subverted). The values may also be provided through an out-of-band process.
    - Use HTTPS to download the information. If the certificates are correct, this verifies that the providing website has the necessary private certifications, and this verification is easy to do. However, the use of HTTPS by itself does not counter subverted websites.
    - Verify that the website and package name is correct. When using a website, temporarily disable internationalized domain names to counter homographic attacks (names that look the same but are not). Consider checking the website's country via the browser, as unexpected results may reveal an attack. Verifying names can help counter typosquatting (where an attacker creates sites or packages with intentionally similar names).
    - Download the information, wait for a period of time, and then verify it again before using it. If the source website is broken into, the website owner may notice and fix it.

- Third party information is retrieved and stored in protected version controlled distribution stores for later distribution (as shown in pink). This ensures, for example, that the information is available even when the external network is under attack. This also makes confidentiality of users easier to achieve, since users typically receive the information indirectly from these stores. This information can be reviewed to determine which information is appropriate for further use.
- An "install/update CE" is created and tools are run to install/update SwA tools so they are prepared for use (shown in the figure as a blue rectangle). In many cases, the CE will not have a connection to an external network (e.g., the Internet); the SwA tools and data would instead be retrieved from the version-controlled repository. However, in some cases an Internet connection may be necessary (e.g., for license key verification or other special purposes). In some enclaves, if an Internet connection is necessary, the "install/update CE" will have to run in an external environment, and the resulting image is separately brought into the environment (the image is not shown in the figure). Images may be tainted, and tainted images cannot be later used by the install/update CE. Note that this is similar to solution 1.
- There are other CEs, each with specific purposes (build, analysis/test/debug, and editing). This is similar to solution 1, but in solution 1 there was just a special CE for each analysis tool. For example, users using the editing environment can create copies of custom code, modify it, check in the modified versions, and request building and testing. Compilation and debugging should occur outside the editing environment where practical, say by allowing debug interaction with isolated test environments. That way, even compromised builds cannot modify code in the editing environment. Here, the approach of using more CEs is expanded to other tools as well, again to limit the privileges to only those needed by the tools.
- Custom code is under VC and edited via an editing CE. Custom code may be a small or large part of the overall system. Many systems today involve mostly third party code and a small amount of integration code. However, even if it is a small part of the overall system, custom code for a given system is critical, so we emphasize it in the figure with a separate color. In practice, the VC system used for custom code may be the same as for other data. In many cases, keeping custom code confidential is vitally important; in other cases, it may be publicly viewable. In all cases, the integrity of custom code must be preserved. Changes to custom code typically should include information on who made the change, the change made, and when the change was made, and ensure that this cannot be forged.

- A logging/monitoring system is included so that malicious activity may in some cases be detected, and in any case those activities can be forensically reviewed later. Logs must typically be protected from later modification (this is often done by using a separate logging system with append-only mechanisms often connected through encrypted channels and sometimes extended with digital signatures or hashes). In practice there must be automated log analysis, because the volume of logs in modern systems is typically too great to only do manually. These may be implemented, at least in part, through an intrusion detection system (IDS) and/or intrusion prevention system (IPS). Application-aware sensors could be added to check changes in application behavior or privileges.[16]
- Honeypots are data or systems that appear legitimate, but are isolated and monitored, and are intended to be appealing to attackers. Properly implemented, they can aid logging/monitoring systems. For example, they can signal real attacks, countering the false positive problems of logging/monitoring systems. Honeynets are networks of honeypots; in practice, honeynets would be used, but the term "honeypot" is more widely known.
- Diodes, aka one-way communication paths, prevent information or attacks from leaking in the reverse direction of a communication path. This prevents certain kinds of exfiltration and is a form of least privilege. Diodes are shown as one-way arrows in Figure 6-2.
- Returned data from isolated environments, such as results of SwA tool analyses, are filtered and isolated for safety based on the anticipated level of risk. This expands on solution 1's method of handling results, as discussed in section 6.A.3, to cover other kinds of results. This filtering and isolation makes it difficult for an attacker to attack other parts of the SDE by creating malicious returned data.

As noted earlier, handling of returned data may be implemented in a variety of ways (e.g., only presenting screen views of returned data, or providing the returned data to a web browser as untrusted service (using the web browser's protection mechanisms to protect the rest of the SDE)). The mechanisms that should be used will depend on the risks (probability and impact of attack), and must consider the measures to be countered. For example:

- Malicious SwA tools may omit warnings of dangerous constructs. This can be countered by using other (additional) tools to detect those constructs.

[16] For example, the OWASP AppSensor project "defines a conceptual framework and methodology that offers prescriptive guidance to implement intrusion detection and automated response into applications."

- Malicious SwA tools may propose obviously malicious changes (e.g., backdoors or disabled functionality). These can be countered by human (manual) review and, in some cases, tool review.
- Malicious SwA tools may propose maliciously misleading code (aka underhanded code). Maliciously misleading code is code that is written to look like it does one thing to human reviewers but instead intentionally does something else.[17] Maliciously misleading code can often be countered with simple countermeasures, such as forcing code reformatting, requiring the use of style checkers, using editor syntax highlighting (which can reveal anomalies such as code hidden in comments), and using memory-safe languages. Reviewers could be required to retype proposed changes to counter problems such as homograph attacks (where similar or identical symbols have unexpected meanings, e.g., swapping lower case "l" with digit "1" or using Cyrillic when Latin characters were expected). Reviews of the results by other tools can also help, since some of those changes may attempt to insert vulnerabilities that other tools can detect.

There are various enhancements that should be done but are not directly shown in the figure:

- Users often need access to other capabilities (e.g., email, external Internet access, etc.). These would typically be provided by still more CEs, which are isolated (at least partially) from the other CEs shown here. That way, for example, a subversion of one tool (such as of an email reader or web browser) does not easily turn into a subversion of the rest of the SDE. This depends on how well the different CEs are isolated.
- As part of SCRM, third party tools, other code, and data are archived and are reviewed before use depending on their risk. For example, malware detectors are used to detect previously identified malicious code. We assume the SCRM process also reviews the supplier (e.g., the supplier's processes, tools, and trustworthiness). This is not as obvious from the figure because the figure shows an architectural view rather than a process view.
- VC software and processes are enhanced, for example, to ensure that what is released is what was reviewed.
- The SDE's environment is hardened against attack and is routinely updated. Access controls are put in place to prevent unauthorized activity.

[17] The Underhanded C Contest <http://www.underhanded-c.org/> has examples of maliciously misleading code.

It is important that these capabilities be easy to use and understand. Users will try to work around mechanisms that are cumbersome to use. Where possible, the mechanisms should be essentially invisible to their users.

2. Meeting Security Goals for Solution 2

Table 6-2 shows how this solution provides countermeasures against various threat agents to meet security goals. In some cells, we put "Cell above" to show a repetition of the cell above it (to reduce the size of the table). We put "Solution 1" to indicate that the corresponding countermeasures of solution 1 also apply.

Table 6-2. Security Goal Coverage by Solution 2

Security Goal/ Threat Agent	Threat Purpose	Confidentiality	Integrity	Availability	Non-repudiation
Outsider	*Intentional*	Encrypted links outside to all suppliers (including tools and code), firewall, logging/monitoring, honeynets, isolation, secured systems	Encrypted links outside, firewall, logging/monitoring, honeynets, verification measures such as digital signatures, secured systems	Encrypted links outside, firewall, redundant links for failover, logging/monitoring, honeynets, secured systems	Logging/monitoring, digital signatures, secured systems
Supplier tool	*Unintentional*	Solution 1 + anonymizer and protected distribution stores	Solution 1 + verification measures such as digital signatures, build/test isolated from development (in addition to tools); version control; results separated	Solution 1 + build/test isolated from development (in addition to tools); version control; results separated	Solution 1 + Tools must digitally signed and verified
	Intentional	Cell above	Cell above + results separated to prevent attack via results, VC requires developers to actively sign changes	Cell above + separated to prevent attack via results	Cell above
Supplier code/data	*Unintentional*	Solution 1 + Code: Execution of third-party software often cannot send data outside (including to supplier) due to isolation; Id:	Solution 1 + Build/test isolated from development; version control; results separated; SCRM (including analysis of supplied items using isolated	Solution 1 + Build/test isolated from development; version control; results separated	Code must be digitally signed and verified

Security Goal/ Threat Agent	Threat Purpose	Confidentiality	Integrity	Availability	Non-repudiation
		anonymizer and protected distribution stores	component test environment)		
	Intentional	Solution 1	Cell above + SCRM analyses focused on looking for intentional (malicious) information	Cell above + separated to prevent attack via results	Cell above
Authorized developer	*Unintentional*	Solution 1 + Code: VC limits who can read; system development, build, and test environments isolated from Internet (no easy exfiltration); diodes prevent info release back; separate environments for Internet access; spillage of identities prevented by anonymizer and protected distribution stores	Solution 1 + VC limits who can make changes; isolation limits damage/unintentional change	Solution 1 + Isolation limits unintentional destruction; backups	VC records who did what change when in NR way
	Intentional	Cell above + logging/monitoring system	Cell above + review, VC records who did what change when and that what's used is what's reviewed	Cell above + ensure developers can't destroy mechanisms (VC, backups, etc.)	Cell above

Of course, other solutions are possible, depending on the risks and costs. See section 5.C for more discussion about designing the SDE for security goals and threats.

One extension to solution 2 would be to add a requirement to have a "reproducible build," aka a "deterministic compilation." "A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts." [Reproducible] The purpose of reproducible builds is to resist attacks where a built artifact does not match its purported source code. More information on this can be found at the reproducible builds site <https://reproducible-builds.org/>.

Another extension would be to do more evaluation of SwA tools and code. We already emphasize doing some examination of third-party SwA tools and code. We could go further and require that SwA tool and code must be examined using detonation chambers before they are used. A detonation chamber is a highly monitored environment specifically designed to trigger, detect, and analyze malicious indicators. Examples of tools that can implement detonation chambers (and sometimes other capabilities) include the following: Comodo Internet Security' Defense+ Sandbox [Comodo], FireEye Malware Analysis, Symantec Content Analysis, and Cuckoo Sandbox. Detonation chambers often require significant resources to maximize detection and analysis. Where used, they should be similar enough to the final environment so that malicious code cannot easily avoid being triggered in them. Solution 2 uses isolated environments to limit damage, but no isolation mechanism is perfect; adding detonation chambers increases the likelihood that malicious code will be detected.

7. Conclusions

In practice, a suite of many SwA tools is necessary to detect vulnerabilities adequately enough to achieve good SwA. However, simply adding them to an unchanged SDE may introduce, or appear to introduce, unnecessary risk. As shown in Chapter 2, there really are adversaries who attack SDEs. Adversaries know that a suite of SwA tools is necessary for higher assurance software, and therefore may increasingly try to attack our systems through our SwA tools.

Our goal in this paper is to help ease the deployment of SwA tools, by countering a potential objection to using them. Instead of being reluctant to use SwA tools, or using them without considering risks, we have shown that it is possible to use many SwA tools while modifying SDEs to manage risks.

In particular, in our "medium protection" sample solution, we have shown that simple steps can reduce risks while using many SwA tools. This sample solution reduces risks by using isolation mechanisms to separate environments based on the task to be done (install/update and analysis). This sample solution can be automated, and in some circumstances it may reduce risk in a relatively uncomplicated manner. These automations could be implemented with simple scripts that are shared widely, making the approach easy to implement.

We recommend that organizations fully embrace the use of many SwA tools when developing software. Where appropriate, they should consider taking the additional steps discussed here if they determine that the risks of using SwA tools are otherwise too high. Our hope is that this information will lead to the widespread safe use of suites of SwA tools.

Appendix A
Proofs of Concept

Here, we provide brief proofs of concept. We first demonstrate the "medium protection" approach using a virtual machine (VM), as implemented by VirtualBox. We then demonstrate the "medium protection" approach using a container, as implemented by Docker. These are essentially experiments to demonstrate that it possible to do these things using either VMs or containers. For the purpose of these experiments, we will assume that the implementation of VMs and containers is trustworthy and provides adequate protection against attack. Note that cloud environments typically employ isolation mechanisms such as containers or virtual machines to separate user data and processing.

These proofs of concept are simple examples. Real use of these approaches would typically be hardened further, and fully automated through some script to make it easy to use them correctly. Those automated scripts would be need to be maintained (possibly by those who maintain the build scripts) and would need to be themselves protected from attack.

A.1. Medium Protection Using a Virtual Machine

Purpose and/or Objective: Demonstrate the "Medium Protection" approach (install, analysis, update) using a virtual machine.

Method or Procedure: Here we installed a VM with a software assurance (SwA) tool, performed an analysis, and then updated the tool, all within a virtual machine with various security restrictions.

We used VirtualBox as a demonstration platform, as it is freely available, available on many systems, and is open source software (OSS) (enabling easy repetition and system changes if necessary). We used a GUI for many actions, as that was convenient; if this was to occur often, those should be automated (VirtualBox supports automation with a large set of script interfaces).

For our experiment, VirtualBox version 5.2.8 on Windows 7 was used as a host with Ubuntu Linux as the guest. We used the OSS tool "flawfinder" as our sample SwA tool; this avoided any possible licensing issue because it is OSS, and most of it was developed by David A. Wheeler.

Part 1. Install Initial Version of the SwA Tool

VirtualBox was installed and started up. A new VM was created via File/New, giving the VM the name "tools," type "Linux," and version "Ubuntu 64-bit."[18] We picked a memory size (8GB) and created a hard disk. For our tests, we created a VirtualBox Disk Image (VDI) image, dynamically allocated, with 100GB. Unless otherwise noted, we kept default values; in particular, the VM had network access via a NAT during installation of the initial version.

We downloaded Ubuntu Linux from https://www.ubuntu.com/; in our test we used Ubuntu desktop, specifically "ubuntu-16.04.4-desktop-amd64.iso." This image has much more than we really need for the experiment (e.g., a web browser and office suite), but it is fine for the experiment. This image included Python3, a language system required for the tool we were about to install. We then right-clicked on the new "tools" VM, selected Settings / Storage, selected the fixed image, chose a virtual optical disk file, set it to the downloaded Ubuntu file, and selected "OK."

We then pressed "Start" to start the new VM. At this point, the usual Ubuntu install began. We selected "Install Ubuntu" and selected "Download updates while installing Ubuntu," "Erase disk and install Ubuntu," and "Continue." We selected our time zone and language, and entered "who are you" information (including a password). We selected "log in automatically," since this VM can only be started by those already logged in. The system then began installing as usual. This took about 20 minutes (this time varies greatly depending on factors such as system and network speed). Once done, we selected "Restart Now." The virtual DVD was automatically ejected, so we confirmed the restart with "Enter" and then had a VM with an installed operating system.

We wanted to be able to access the system using secure shell (SSH), so we installed the SSH server. We also needed to install pip, an installer for software that uses Python3. We first right-clicked in the background and selected "Open Terminal." We then installed it using the following (the first install requires the user password, and the "--yes" option automatically downloads and installs dependencies):

```
sudo apt-get install --yes openssh-server
sudo apt-get install --yes python3-pip
```

We then installed a specific version of a SwA tool (in this case, flawfinder version 2.0.5) via the terminal. We downloaded and installed it using the following command (note there are two equal signs, and we use pip3 so that the tool will use Python3)[19]:

---

[18] The ending periods and commas here are in most cases not part of the data. IDA's technical style requires that we insert periods and commas within quotation marks, even when they are not part of the data.

[19] It is possible to tell pip3 to ignore certificates by passing it the option and argument "--trusted-host pypi.python.org" just after "install." For experimentation, we used this approach instead of installing

```
pip3 install flawfinder==2.0.5
```

We then cleanly shut down the virtual machine (you can do this by right clicking on the star on the right-hand side and then selecting “Shut down”).

We then had an untainted image with a working environment, including the SwA tool. However, we didn’t want to run any analysis directly in this working environment, because we will want to later update it with access to the network, and when network access is enabled we want to be confident that the system running this image uses an untainted image (an image that cannot have the analyzed material).

There are a large number of different ways to implement this requirement. We could have used VirtualBox’s ability to create snapshots and use differencing images (which can be chained), so that when we ran an analysis, all changes would go to a separate real file that could later be deleted. This might be especially important to do with some proprietary software, since in some cases the license may forbid making a copy (even if it is not being used). This is not relevant in this example; all of the software used in this experiment is OSS and thus may be freely copied.

To keep things simple, we simply created a pristine backup copy of the storage image. This copy remained untainted. We made this copy by first exiting the VirtualBox application (to ensure that the running application didn’t interfere with anything) and then executing these Windows commands (where “dwheeler” is the current user) to copy all of the state of that machine (contained in the “tools” directory, including its configuration file tools.vbox and its disk image tools.vdi):

```
cd \Users\dwheeler\VirtualBox VMs
robocopy tools tools.PRISTINE /COPY:DATSO /MIR
```

We can later copy the backup (pristine) copy back onto the original “tools.vdi” and thus erase any work done in it since.

The VirtualBox documentation has many warnings about creating clones of disk images that do not apply in this case. It warns about simple file copying like this, and says that copying must instead be done using VirtualBox’s Virtual Media Manager (in \Program Files\Oracle\VirtualBox) because “VirtualBox assigns a unique identity number (UUID) to each disk image, which is also stored inside the image, and VirtualBox will refuse to work with two images that use the same number.” However, we didn’t plan to ever use the images simultaneously, so there was no need to use the Virtual Media Manager in this case. In a more sophisticated setup, the Virtual Media Manager (or similar tool) might be needed.

---

certificates in a temporary VM. In real-world applications, this option should not be used, since it enables a man-in-the-middle attack. Also, executing pip3 in this environment notified us that a more recent version of pip3 was available and how to perform the update. For our purposes, the version of pip3 didn’t matter, so we ignored the notification.

Part 2. Run SwA Tool in a Constrained Environment

With the VM shut down, we went to "Settings/Network" and changed the network settings so that the VM could no longer leak data out to the Internet. The most secure setting would be to completely disable the network, but that would mean that we could not use a virtual network to transfer information to and from it. For our experiment, we selected "host-only adapter," so that the VM can contact the host machine over the network but *not* the outside network. This is not the only approach. For example, we could have modified the image to place files within them for analysis. But the approach described here is enough for demonstration purposes.

We then selected the VM, selected "Start" to run the VM, and again right-selected the background to "Open Terminal." We created a directory to work in and ran "ifconfig" inside the VM to find its non-loopback IP address (192.168.56.101 in our case).

```
mkdir analysis; cd analysis
ifconfig
```

We then needed to transfer information for the tool to analyze. We analyzed the trivial test file "junk.c" provided in flawfinder version 2.0.5 (version 2.0.6 has the same unchanged file). We separately downloaded that into the host system and sent it to the guest system using the usual "secure copy" tool (which runs over SSH):

```
scp -r junk.c dwheeler@192.168.56.101:/home/dwheeler/analysis/
```

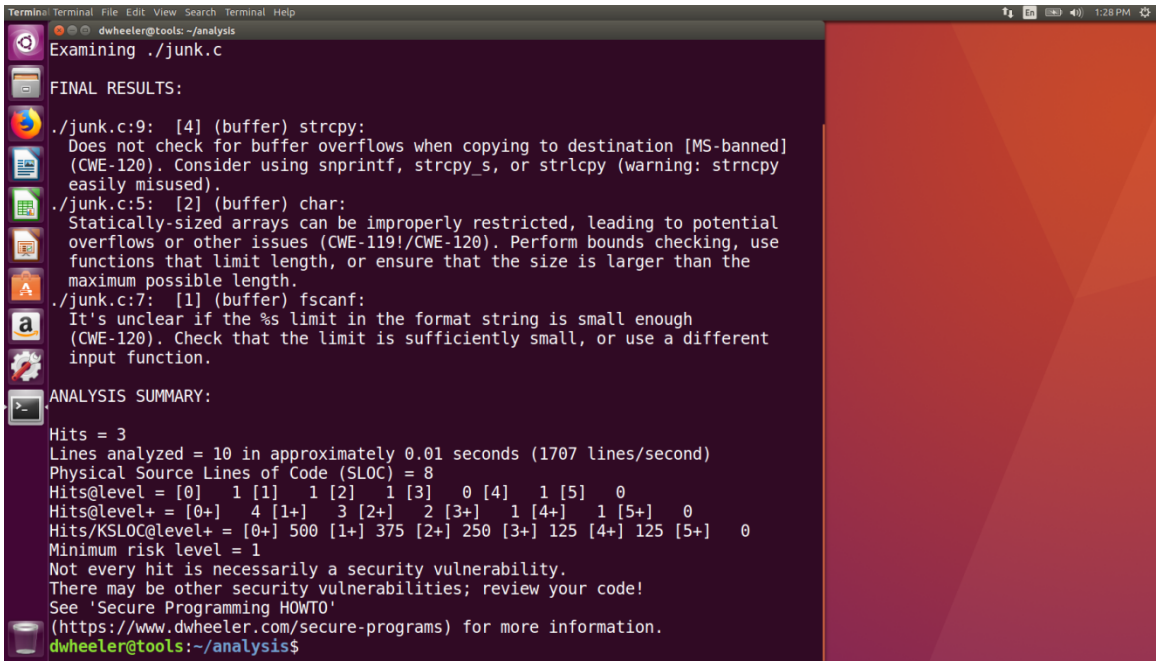
We confirmed connection, provided the password, and the file for analysis was copied. The "-r" option isn't really necessary here, but that would allow recursive copying of whole directories (not just a single file). Other tools, such as rsync, could also be used to copy this data for analysis. (We didn't use rsync, since it's not needed and we would have had to install it.)

Note that at this point, this image became tainted, because it had potential access to the source code. In this case, the source code is actually in the image.

Now, on the VM side, we could run the SwA tool to perform analysis. Here, we ran the tool in a simple way (the "." means "begin at current directory"):

```
flawfinder ./
```

The following figure shows a screenshot after the SwA tool has executed on the test software (after making it full-screen and changing the font size to 18 point). The details of the display aren't important; what's important is that the SwA tool was run and produced analysis results.



Here, we are depending on the virtual machine monitor (VMM) to prevent the display of the SwA tool results from being able to attack the larger system.

Once we were done, we shut down the guest VM.

Part 3. Update SwA Tool

Now we want to upgrade the SwA tool. We cannot upgrade the image we just used, because it is tainted. However, we can replace the tainted image with a copy of an untainted image. Here is how we did that. We first shut down VirtualBox. On the host command line, we overwrote the modified image and restored the original backup ("pristine") version:

```
cd \Users\dwheeler\VirtualBox VMs
rem We can keep modified version by running rename tools tools-modified
robocopy tools.PRISTINE tools /COPY:DATSO /MIR
```

At this point, the tainted image has been erased and been replaced with an untainted image.

We then started up VirtualBox. We checked to confirm that the VM configuration once again had access to the network (Settings/Network are NAT). However, this image has never had access to the software that was analyzed, so the update process cannot leak any information about the software that was analyzed.

We then selected "Start," right-clicked in the background, and opened a terminal.

Next, we needed to update the SwA tool. This was done with this command:

```
pip3 install --update flawfinder==2.0.6
```

Then we shut down the VM. Once again, we could stop VirtualBox and back up the state of the system to create a pristine version for future updates.

Conclusion or Discussion

In short, we were able to implement the medium protection approach using VMs. In practice, it would be important to automate this process (to simplify its use and avoid errors from incorrect application). There are many other ways to implement this approach using VMs; this is merely an example to illustrate the approach.

A.2. Medium Protection Using a Container

Purpose and/or Objective: Demonstrate the "Medium Protection" approach (install, analysis, update) using a container.

Method or Procedure: Here, we created a container image that includes a SwA tool of a particular version, performed an analysis, and then created a new container image that contained a newer version of the SwA tool.

We used Docker as a demonstration platform, as it is freely available, available on many systems, and is OSS (enabling easy repetition and system changes if necessary). We ran Docker on top of Ubuntu, which in turn was within a virtual machine implemented by VirtualBox (as described above). For our experiment we used Docker version 18.03.0-ce build 0520e24, running inside Ubuntu 16.04.4 (ubuntu-16.04.4-desktop-amd64.iso). Ubuntu was running within VirtualBox version 5.2.8 while Windows 7 was used as a host. We again used the OSS tool "flawfinder" as our sample SwA tool; this avoided any possible licensing issue because it is OSS and most of it was developed by David A. Wheeler.

In practice, most Docker commands require running root. We're not thrilled about this requirement; one solution is to run Docker within a virtual machine to isolate the containers further. However, this was not a problem for our demonstration.

Docker uses a "Dockerfile" to create container images. The following Dockerfile was developed to create container images for our experiment.

```
# Dockerfile: Create an image for flawfinder analysis that is restricted
# in what it has access to.
#
FROM python:3.6.5-slim-stretch
ARG FLAWFINDER_VERSION
#
# We could create a special unprivileged user inside this container, but
# the user is actually unprivileged to start with, so we won't bother.
#
```

```
    # This uses "pip3" to install flawfinder (per its directions).
    # If the host system is Ubuntu, pip3 requires disabling its default DNS masking.
    # Do this by editing /etc/NetworkManager/NetworkManager.conf
    # to comment out or remove this line:
    # dns=dnsmasq
    # For more information, see:
    #  https://stackoverflow.com/questions/44761246/temporary-failure-in-name-resolution-
errno-3-with-docker
    #
    # On our system must disable certificate checking or provide certs.
    # For our experiment, we'll disable using --trusted-host. Don't do this
    # in 'real' systems.
    #
    RUN pip3 install --trusted-host pypi.python.org flawfinder==${FLAWFINDER_VERSION}
    #
    # Instead of providing a fixed "entrypoint" in the build, we'll let users
    # provide the command. This is much more flexible.
    dwheeler@tools:~/demo$ cat Dockerfile
    # Dockerfile: Create an image for flawfinder analysis that is restricted
    # in what it has access to.
    #
    FROM python:3.6.5-slim-stretch
    ARG FLAWFINDER_VERSION
    #
    # We could create a special unprivileged user inside this container, but
    # the user is actually unprivileged to start with, so we won't bother.
    #
    # This uses "pip3" to install flawfinder (per its directions).
    # If the host system is Ubuntu, pip3 requires disabling its default DNS masking.
    # Do this by editing /etc/NetworkManager/NetworkManager.conf
    # to comment out or remove this line:
    # dns=dnsmasq
    # For more information, see:
    #  https://stackoverflow.com/questions/44761246/temporary-failure-in-name-resolution-
errno-3-with-docker
    #
    # On our system must disable certificate checking or provide certs.
    # For our experiment, we'll disable using --trusted-host. Don't do this
    # in 'real' systems.
    #
    RUN pip3 install --trusted-host pypi.python.org flawfinder==${FLAWFINDER_VERSION}
    #
    # Instead of providing a fixed "entrypoint" in the build, we'll let users
```

```
# provide the command. This is much more flexible.
```

It is a good practice to create a non-privileged user within the container, and then use that. We did not bother to do that here, since this is a simple experiment, but that should be done if the system is to be seriously used.

We also created a small “makefile” to store the scripts used to implement the experiment as follows (the long indents are initial tab characters):

```
# Demo use of Docker to contain a potentially-malicious SwA tool.
#
# Here we set the default version of the SwA tool.
# This can be overridden, e.g.:
# make FLAWFINDER_VERSION=2.0.6 run
FLAWFINDER_VERSION=2.0.5

# The "run" command invokes copy-analysis to create an analysis directory,
# and then runs the Docker image to perform the analysis.
# We make a copy *and* do a readonly bind mount to ensure that the SwA tool
# cannot modify the "real" files (either one would be enough, but using
# both approaches makes it even harder to thwart).
#
# We accept data *back* from the tool... but what if the tool is malicious?
# To deal with that, we filter the output against a whitelist to prevent
# attacks. In our case, we use "tr" to implement a whitelist of
# the set of permitted characters (this case, we only allow tab,
# return, newline, and printable ASCII). This means that anything not listed
# (such as control characters) will be filtered out.
# The result, after filtering, is stored in "results".

run: copy-analysis
 sudo docker run --network none --read-only --tmpfs /tmp --tmpfs /tmp \
 --mount type=bind,source="${PWD}/analyze",destination=/mnt,readonly \
 -w /mnt \
 flawfinder-demo-$(FLAWFINDER_VERSION) flawfinder ./ | \
 tr -cd '\t\r\n[:print:]' > results

# Copy files to be analyzed into a separate directory.
# We create copies so that the SwA tool never has an opportunity to edit
# the 'real' files, and thus cannot modify anything in an unauthorized way.
```

```
copy-analysis:
 rm -fr analyze/
 mkdir analyze/
 cp -p *.c analyze/ # do whatever to copy files into analyze/

# The "build" command builds a Docker image from its Dockerfile.

build:
 sudo docker build -t flawfinder-demo-$(FLAWFINDER_VERSION) \
 --build-arg FLAWFINDER_VERSION=$(FLAWFINDER_VERSION) .

.PHONY: run copy-analysis build
```

Part 1. Install Initial Version of the SwA Tool

We created a container with the SwA tool on a system connected to the network by running the following command:

```
make build
```

This is an extremely simple command, because the real work is done by the files listed earlier. This command kicks off a “docker build” command, which downloaded the necessary images and ran the installation commands to create a Docker container. Our makefile by default installed flawfinder 2.0.5, so this command created a container named flawfinder-demo-2.0.5. The resulting image was untainted, since it never had access to the data to be analyzed that should not be publicly released.

Part 2. Run SwA Tool in Constrained Environment

We ran the container to analyze software by doing the following:

```
make run
```

This copied files to be analyzed into a special “analysis” directory that is outside the container image and then ran the container using the previously created container image to analyze those files. Note that in this step we expressly disable access all network access by the container. All filesystems here have been made read-only or temporary. A “bind mount” is used to give the container temporary access to a copy of the source code. Since the image with the SwA tool is always run read-only, it cannot become tainted.

As the results are retrieved, we pass the results through a whitelisting process. In this particular case, we only accept the following characters: tab, carriage return, newline, and printable characters. The results, after filtering, are stored in the file “results.” This means that even if the SwA tool attempts to create malicious output (e.g., by inserting escape codes or other control information), those results are first filtered to prevent many kinds of attacks.

Part 3. Update SwA Tool

We then needed to demonstrate upgrading the SwA tool. We found it easier to simply create a new container, which we did as follows:

```
make FLAWFINDER_BUILD=2.0.6 build
```

Since this builds a new container from scratch, there is never an opportunity for the SwA tool to reveal any data from previous analyses – the new container has never had an opportunity to see that data.

Once the new container is built, it can be used. For this example, we executed it this way:

```
make FLAWFINDER_BUILD=2.0.6 run
```

Conclusion or Discussion

In short, we were able to implement the medium protection approach using containers (as well as VMs). In practice, it would be important to automate this process (to simplify its use and avoid errors from incorrect application). There are many other ways to implement this approach using containers; this is merely an example to illustrate the approach.

Appendix B
Medium Solution: Quick Implementation Guide

This appendix provides a quick technology-independent summary of the key requirements of solution 1 (medium protection) described in section A. The purpose of this approach is to reduce the risk that a software assurance (SwA) tool that is vulnerable (intentionally or not) is unlikely to lead to the loss of confidentiality, integrity, or availability.

The process for executing the SwA tool needs to implement two computing environments (CE):

1. *Install/update*. In this CE, the tool and its dependencies are installed and may be updated, and network access would typically be allowed, but there is never read or write access to non-public data (including source code). This CE creates an untainted image that will be used in the analysis CE; only untainted images can be connected to the install/update CE.
2. *Analysis*. In this CE, the tool is used for analysis, so it must have read access to non-public data (such as source code). It may be able to write analysis reports, but the system protects from potential attacks via those reports (e.g., the report may only be accepted if it is in an intentionally limited format, it may go through input validation or filtering to counter potential attack, or trusted tools such as web browsers may be used that are designed to resist attacks). The tool running in this analysis CE may be able to modify copies of source code as proposed changes, but those proposed changes can be reviewed before they are accepted. The tool cannot share non-public data to outsiders or other untrusted users (e.g., it has no external network connection, and any execution cannot leak its data later to untrusted users). The analysis CE uses an image created by the install/update CE, which may be read-only or allow writes:
    a. If the image is always read-only while in analysis CE, then the image can be reused because non-public data cannot leak into the read-only image. This is an “untainted” image.
    b. The analysis CE may instead allow writes to the image while in analysis CE. If writes are allowed, that modified image must never be used in install/update CE, because the modified image may include non-public data that could be leaked to an external network. This is a “tainted” image.

The SwA tool must be isolated in both CEs so that it cannot "break out" of the CE isolation mechanisms. These isolation mechanisms must be adequately strong to defend against the expected level of attack. For examples of how to implement this approach using virtual machines and containers, see Appendix A.

Appendix C
DoD Policies on Countering Supply Chain or Software Development Environment (SDE) Attacks

DoD has a number of security policies, many of which relate to countering attacks on the software supply chain and/or SDEs. This section discusses some DoD policies related to them, since the DoD has its own complexities and must counter determined adversaries.

A.1. DoDI 5000.02

DoD Instruction (DoDI) 5000.02 (*Operation of the Defense Acquisition System*) "assigns, reinforces, and prescribes procedures for acquisition responsibilities related to cybersecurity in the Defense Acquisition System." Its enclosure 14, "Cybersecurity in the Defense Acquisition System," notes that "Cybersecurity is a requirement for all DoD programs and must be fully considered and "implemented in all aspects of acquisition programs across the life cycle." Enclosure 14 states that the system architecture and design will address how the system "is structured to protect and preserve system functions or resources, e.g., through segmentation, separation, isolation, or partitioning" and "is configured to minimize exposure of vulnerabilities that could impact the mission, including through techniques such as design choice, component choice, security technical implementation guides, and patch management in the development environment (including integration and T&E), in production and throughout sustainment" [DoDI 5000.02]. Enclosure 14 also states that Program Managers will "incorporate automated software vulnerability analysis tools throughout the life cycle to evaluate software vulnerabilities, as required by Section 933 of Public Law 112-239."

DoDI 5000.02 enclosure 3 includes a discussion of the program protection plan (PPP). It notes that "where a DoD capability advantage derives from the integration of commercially available or custom-developed components, program protection manages the risk that design vulnerabilities or supply chains will be exploited to destroy, modify, or exfiltrate critical data, degrade system performance, or decrease confidence in a system." Enclosure 3 also states the following:

> "Program managers will describe in their PPP the program's critical program information and mission-critical functions and components; the threats to and vulnerabilities of these items; the plan to apply

countermeasures to mitigate associated risks; and planning for exportability and potential foreign involvement. Countermeasures should include anti-tamper, exportability features, security (including cybersecurity, operations security, information security, personnel security, and physical security), secure system design, supply chain risk management, software assurance, anti-counterfeit practices, procurement strategies, and other mitigations in accordance with DoD Instruction 5200.39 (Reference (ai)), DoD Instruction 5200.44 (Reference (aj)), and DoD Instruction 8500.01 (Reference (x)). Program managers will submit the program's Cybersecurity Strategy as part of every PPP. Countermeasures should mitigate or remediate vulnerabilities throughout the product life cycle, including design, development, developmental and operational testing, operations, sustainment, and disposal. Program Managers will implement the use of automated software vulnerability detection and analysis tools and ensure risk-based remediation of software vulnerabilities is addressed in PPPs, included in contract requirements, and verified through continued use of such tools and testing (as required by section 933 of P.L. 112-239, Reference (l))." [DoDI 5000.02]

A.2. Program Protection Plan (PPP)

To implement a PPP, the DoD provides a PPP outline and guidance [DoD PPP]. This guidance recommends that programs address questions related to the development environment such as:

- "How will the program identify new vulnerabilities (both system-level and in the development environment) to the [Critical Program Information (CPI)] and mission-critical functions and components?"
- "Indicate the RFP Contract Line Item Number (CLIN) or Data Item Description (DID) that will be used to ensure that CPI and critical functions/components are protected in the development environment and on the system"
- "Contractor development environments may host CPI and should be evaluated for protection."
- "How will the development environment be protected? / List the development environment tools"
- Who has access to the development environment?

In addition, the supporting PPP template [DoD PPP Template] asks additional questions about the development environment:

- "How will software architectures, environments, designs, and code be evaluated with respect to CVE (Common Vulnerabilities and Exposures), CAPEC (Common Attack Pattern Enumeration and Classification), and CWE (Common Weakness Enumeration)?"

- "Explain how supply chain threat assessments will be used to influence system design, development environment, and procurement practices. Who has this responsibility? When will threat assessments be requested?"
- "Specify the way in which the program will identify new vulnerabilities to the CPI and mission-critical functions and components (both system-level and in the development environment)."

In short, the PPP outline and guidance, along with its supporting template, clearly notes that the development environment is important. However, their primary suggested approaches focus on areas such as carefully selecting tools and having controlled access with only cleared personnel. If the tools themselves may be malicious, these areas may not be enough. As we will discuss later, we offer approaches to protecting the development environment even if malicious tools slip through evaluation and end up being used.

A.3. DoDI 5200.44

DoD Instruction 5200.44, *Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN),* "establishes policy and assigns responsibilities to minimize the risk that DoD's warfighting mission capability will be impaired due to vulnerabilities in system design or sabotage or subversion of a system's mission critical functions or critical components, as defined in this Instruction, by foreign intelligence, terrorists, or other hostile elements." [DoD 5200.44]

This instruction states that "Risk to the trust in applicable systems shall be managed throughout the entire system lifecycle. The application of risk management practices shall begin during the design of applicable systems and prior to the acquisition of critical components or their integration within applicable systems, whether acquired through a commodity purchase, system acquisition, or sustainment process."

It states that for applicable systems risk management shall include processes, tools, and techniques such as:

- "Reduce vulnerabilities in the system design through system security engineering."
- "Control the quality, configuration, software patch management, and security of software, firmware, hardware, and systems throughout their lifecycles…."
- "Detect the occurrence of, reduce the likelihood of, and mitigate the consequences of unknowingly using products containing counterfeit components or malicious functions…."

- "Detect vulnerabilities within custom and commodity hardware and software through rigorous test and evaluation capabilities, including developmental, acceptance, and operational testing."

DoDI 5200.44 includes a reference to the "SCRM Key Practices guide," whose development was led by IDA [Wheeler 2010].

This also references the PPP.

A.4. DoDI 8500.01

DoD Instruction 8500.01, *Cybersecurity*, applies to "All DoD IT" and "All DoD information in electronic format" [DoDI 8500.01]. It states that:

- "DoD will implement a multi-tiered cybersecurity risk management process…."
- "Risks associated with vulnerabilities inherent in IT, global sourcing and distribution, and adversary threats to DoD use of cyberspace must be considered…."
- "Risk management will be addressed as early as possible in the acquisition of IT and in an integrated manner across the IT life cycle."

This instruction briefly discusses enclaves and refers to Committee on National Security Systems (CNSS) Instruction 4009 for their definition. CNSS Instruction 4009 defines enclave as "A set of system resources that operate in the same security domain and that share the protection of a single, common, continuous security perimeter" [CNSSI 4009], a definition from IETF RFC 4949 Version 2.

DoDI 8500.01 does not directly refer to software development environments.

A.5. Risk Management Framework (RMF)

NIST Special Publication 800-39 is intended to provide guidance to the entire U.S. federal government for "an integrated, organization-wide program for managing information security risk to organizational operations (i.e., mission, functions, image, and reputation), organizational assets, individuals, other organizations, and the Nation resulting from the operation and use of federal information systems." It states that senior leaders/executives define the organizational risk frame (tier 1) and that mission/business owners apply their understanding of the organizational risk frame to address concerns specific to the organization's missions/business functions (tier 2). At tier 3, "program managers, information system owners, and common control providers apply their understanding of the organizational risk frame based on how decision makers at Tiers 1 and 2 choose to manage risk," and the Risk Management Framework (RMF) is the primary means for addressing tier 3 risk. [NIST SP 800-39]

NIST Special Publication 800-37 was developed to transform "the traditional Certification and Accreditation (C&A) process into the six-step Risk Management Framework (RMF)." Three of its key steps are to select, implement, and assess security controls. It also states that "Information security requirements are satisfied by the selection of appropriate management, operational, and technical security controls from NIST Special Publication 800-53."

NIST Special Publication 800-53 provides "guidelines for selecting and specifying security controls for organizations and information systems." In particular, it provides a "security controls catalog," a list of many potential security controls organized into families. These families include Configuration Management (CM), Risk Assessment (RA), and System and Services Acquisition (SA). Within the families are specific controls, and some controls have many possible enhancements. NIST SP 800-53 version 4 lists over 800 possible security controls or control enhancements. NIST 800-53 identifies some of these controls or control enhancements as assurance-related controls (i.e., those controls are intended to increase assurance).

NIST SP 800-53's control, SA-12 (Supply Chain Protection), specifically focuses on supply chain and has a number of enhancements. NIST SP 800-53 includes some controls that support the custom development of secure software. These include:

- SA-4 Acquisition Process,
- SA-8 Security Engineering Principles,
- SA-10 Developer Configuration Management,
- SA-11 Developer Security Testing and Evaluation,
- SA-14 Criticality Analysis,
- SA-16 Developer-Provided Training,
- SA-17 Developer Security Architecture and Design,
- SA-20 Customized Development of Critical Components,
- RA-5 Vulnerability Scanning.

NIST SP 800-53's control SA-15 (Development Process, Standards, and Tools) covers software development processes and tools, and includes a number of control enhancements. For example, SA-15(7) (Development Process, Standards, and Tools | Automated Vulnerability Analysis) states that, "The organization requires the developer of the information system, system component, or information system service to: (a) Perform an automated vulnerability analysis…." The development environment is also mentioned in SA-4 (Acquisition Process).

However, the security of the development environment against potentially malicious tools is not especially a strong focus of SP 800-53 version 4. There are certainly controls that describe general measures that can be taken to resist attacks, such as SI-3 (Malicious Code Protection), AC-4 (Information Flow Enforcement), SC-3 (Security Function Isolation), and SC-7 (Boundary Protection). But this is not the same as making it clear that it might be useful to counter attacks from development tools themselves.

No single set of security controls would be appropriate to all systems. NIST SP 800-53 addresses this by identifying baseline controls, which are "the starting point for the security control selection process." NIST SP 800-53 identifies three security control baselines "corresponding to the low-impact, moderate-impact, and high-impact information systems."

DoD Instruction 8510.01, *Risk Management Framework (RMF) for DoD Information Technology (IT),* establishes "the RMF for DoD IT" [DoDI 8510.01]. However, the DoD and intelligence community use a finer-grained approach to selecting controls. As described in Committee on National Security Systems (CNSS) Instruction 1253, system requirements are divided into confidentiality, integrity, and availability (CIA), and the impact is selected as being low, medium, or high for each division. These values then determine the recommended set of baseline controls. [CNSSI 1253]

It is important to understand that the baseline controls are not necessarily implemented by all systems. As NIST SP 800-53 version 4, appendix E explains, "When assurance-related controls cannot be satisfied, organizations can propose compensating controls (e.g., procedural/operational solutions to compensate for insufficient technology-based solutions) or assume a greater degree of risk with regard to the actual security capability achieved."

A.6. Enclave Test and Development (T&D) Security Technical Implementation Guide (STIG)

The T&D STIG is intended to provide the "information protection guidance necessary to ensure secure implementation of Information Systems (ISs) and networks providing test and development services. [They] provide guidance for the separation of network traffic, functionality, and supplement existing security requirements already levied against test and development systems." It defines several kinds of zones, zones A through D, with different required characteristics such as network connectivity and STIG compliance [DISA Enclave 2016]. It defines the zones as follows:

- "The Zone A environment is typically configured as a mirrored operational network for final end stage testing. This environment will have connectivity to the live operational network for final data testing prior to the product or application deployment into the operational network… Development within the

environment should be minimal for final revisions and minor updates of products in the final testing phase. While all systems performing development must be IA compliant, the use of compilers and other development tools on these systems are permitted with approval from the organization's Authorizing Official."

- "Zone B follows and is similar to Zone A from a network connectivity perspective, but with much stricter control mechanisms in the infrastructure supporting the environment. The Zone B environment is the designated zone permitting connectivity for moving sanitized data for testing purposes along with development of applications destined for a live and operational DoD network…. Full development within the environment will be crucial for initial coding and tweaking of products in development phase. While systems performing development must be IA compliant, permitting the use of compilers and other documented development tools on these systems is permissible."
- "Zone C environments are specific in nature to organization's that have a mission to interconnect with other organization's to create a fully closed multi-environment network for product testing and evaluation…. In Zone C, the network will be isolated from the rest of an organization's operational network. Direct access to the DISN is not permitted for Zone C environments as the DISN is used to transmit data between environments."
- The Zone D environment is "a fully closed and physically separate network from any DoD live operational network. Permitted activities in the environment includes, but are not limited to, extensive testing using prohibited tools, working with malicious code, virus samples, working with Ports, Protocols, and Services (PPS) that are otherwise restricted via DoD policy…. Development within the environment is generally not an encouraged practice. If development occurs, all systems performing development must be IA compliant. The use of compilers and other development tools on these systems will be permitted with documented approval from the organization's AO. Prohibiting the connection of development systems within the environment connected to any internal network configured for the environment is required, in particular if Internet access is available. Any applications developed in the environment must be in compliance with the Application Security and Development (ASD) STIG. All applications must go through a code review to ensure the application will not pose a risk to DoD networks when migrated."

Note that development is (mostly) discouraged in Zones A and D, and Zone C is a special case. Thus, of the identified zones, only Zone B allows real development. Yet Zone B has a lot of overhead (e.g., you have to specially review each tool, and it must have an information assurance (IA)-compliant infrastructure). This makes it difficult to

do real work with modern tools. Also, Zone B doesn't protect against potentially malicious tools (because there is no required protection required inside the zone).

The Enclave T&D STIG notes that it's important to protect source code's confidentiality and integrity:

> Building a secure infrastructure will minimize the risk of theft and corruption of source code either accidentally or maliciously. Remote access capabilities described for each zone environment is crucial for testers and developers to access the appropriate tools needed to do their job while maintaining the proper physical and/or logical separation. Development of applications is the most important aspect of the T&D environments. Securing the source code needs to be the highest priority prior to migration into a live operational network. Compromise of the code can cause integrity and availability issues if proper vetting is not complete prior to migration.

This STIG specifically discusses virtualization:

> While implementing virtualized systems into T&D environments to reduce infrastructure costs, security should be a priority to thwart the risk of data theft or other malicious attacks and unintentional activity in the virtualized environment. Virtualizing the T&D environment can be a great way to reduce overall systems and save time in standing up new testing platforms in an ever-growing environment. However, relaxing separation restrictions should be assessed when dealing with different levels of data sensitivity and classification. The most important rule is that no system spanning classifications levels shall be allowed to reside on the same physical host. Securing systems to the highest classification otherwise is necessary, the risk for potential theft and spillage may occur.
>
> Virtualization within differing zones may occur but only if they reside with other like systems. Secure virtualized development systems may not reside on the same physical platform or share the same hypervisor as a non-compliant virtualized testing platform. Zone A and B may be shared across physical hosts if the systems are separated in a systematic manner where proper logical separation is configured and IA-compliant T&D standards are met. All physical hosts running the hypervisor must be IA compliant when connected to any network with outside connectivity. They must be managed through a network segment dedicated for management work only.

However, this STIG focuses on network mechanisms (which may be virtualized in some cases) to provide isolation and protection. This means that there is no necessary protection within an environment should malicious tools be present.

In addition, this STIG has limited applicability. Its scope does not include "networks not directly connected to the Defense Information Systems Network (DISN)," nor does its scope include "Research, development, testing, and evaluation of Platform IT" [DISA

Enclave 2016]. Thus, in many cases, this STIG does not necessarily apply (e.g., in the development of weapon systems), though of course organizations can apply it if they see fit. An easy way to escape all the zone requirements is to develop on a system not connected to the DISN, as this evades the entire STIG. In practice, software development is often done in isolated systems, isolated networks, or in some cases systems connected to the Internet but without access to test data.

A.7. Application Security and Development (ASD) STIG

The ASD STIG is intended to "improve the security of Department of Defense (DoD) information systems," and is "designed to be applied to all enterprise applications connected via the network." It is a "requirement for all DoD- developed, -architected, and -administered enterprise applications and systems connected to DoD networks. An *enterprise application* (EA) is defined as an application or software that is used by the organization to assist in the execution of the organizations missions or meeting organizational goals or tasks." It may be used "for both in-house application development and to assist in the evaluation of the security of third-party applications," however, some sections may not apply to third-party products. [ASD STIG 2018]

The ASD STIG clearly states that tools are important. In section 4, it identifies two kinds of tools in particular along with recommendations for their use:

1. Application Code Scanner. The ASD STIG defines this as "an automated tool that analyzes application source code for security flaws, malicious code, and back doors… These tools can often help developers identify potential flaws in the program logic allowing them to correct the issue prior to application release. Source code is not always required in order to perform code security tests. Some application code scanners will operate on binary or compiled byte code allowing system administrators to perform code scanning tests on application code without having access to the actual source code itself." The ASD STIG states that "Application code scanners should be utilized whenever [possible, particularly] in the development environment where code that has been identified as requiring remediation can be addressed prior to release."
2. Application Scanner (active vulnerabilities testing tool). The ASD STIG defines this as "a tool that is able to communicate with the application and test the application for known security vulnerabilities. An application scanner can be used to test development or production application systems for security vulnerabilities resulting from either application code errors or application system misconfigurations. These vulnerabilities include SQL Injection, Code Injection, Cross Site Scripting (XSS), file disclosures, permissions, and other security vulnerabilities that can be found in network accessible applications. Application vulnerability scanners can identify security weaknesses that are

related to the underlying system configuration enabling administrators to reconfigure systems in order to eliminate identified vulnerabilities." The ASD STIG states that "Application vulnerability scans must be utilized and should be conducted on a regular basis, such as after any product updates or major reconfigurations and prior to activating new applications in their production environment."

The ASD STIG defines severity category codes (each referred to as a CAT) as "a measure of vulnerabilities used to assess a facility or system security posture." Each security rule is assigned a severity category code of CAT I, II, or III. CAT I is the most "Any vulnerability, the exploitation of which will directly and immediately result in loss of Confidentiality, Availability, or Integrity."

The ASD STIG contents list a large number of rules. For example, rule id "SV-84899r1_rule" is titled "The application must not be vulnerable to overflow attacks" and provides the following discussion: "A buffer overflow occurs when a program exceeds the amount of data allocated to a buffer. The buffer is a sequential section of memory and when the data is written outside the memory bounds, the program can crash or malicious code can be executed…."

A positive trait of the ASD STIG is that it can be placed on contracts, unlike most guidance documents that are not designed for that purpose. It also encourages some automation, which is important for modern software.

However, the ASD STIG rules have many gaps. For example:

- Rule "SV-84899r1_rule" for overflow attacks only directly discusses writing outside a buffer boundary. Some attacks, such as Heartbleed, involve reading outside a buffer boundary [Wheeler 2017Heartbleed], a possibility not considered in the ASD STIG.
- Rule "SV-84865r1_rule" mentions XML External Entity (XXE), but only states that, "An XML firewall function must be deployed to protect web services when exposed to untrusted networks." This is weak, because no other countermeasures for XXE are suggested, and there is no reason to believe this would always be sufficient. What's more, this is only given CAT II severity (less important). This is in contrast to the 2017 edition of the Open Web Application Security Project (OWASP) top 10 [OWASP 2017]. This widely used publication added the need to address XXE attacks to their list of the most important attacks to address due to their increasing prevalence. Such attacks could have a variety of impacts on an SDE.

Another problem with the ASD STIG is that it maps rules to specific severity categories. This is a simplifying assumption, but it is often wrong. In the end, severity

should be mapped to mission needs, not just to a technical type of vulnerability. Trained developers or evaluators should be able to propose alternative priorities based on mission need (e.g., because a given vulnerability is not exploitable, is exploitable only by trusted administrators and thus less important, or is more important in a given context).

Appendix D
Acronyms and Abbreviations

AO	Authorizing Official [DISA Enclave 2016]
APL	Approved Products List
ASD	Application Security and Development
BSIMM	Building Security in Maturity Model
CAPEC	Common Attack Pattern Enumeration & Classification
CE	Computing Environment
CIA	Confidentiality, Integrity, and Availability
CII	(Linux Foundation) Core Infrastructure Initiative
CLIN	(RFP) Contract Line Item Number
CM	Configuration Management
CNSS	Committee on National Security Systems
CoN	(Army) Certificate of Networthiness
COTS	Commercial Off-the-Shelf
CPI	Critical Program Information
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
CWRAF	Common Weakness Risk Analysis Framework
CWSS	Common Weakness Scoring System
DADMS	Department of the Navy Application and Database Management System
DID	Data Item Description
DISA	Defense Information Systems Agency
DISN	Defense Information Systems Network
DoD	Department of Defense
DoDI	DoD Instruction

DODIN	DoD Information Network (DISA)
DON	Department of the Navy
EA	Enterprise Application
GOTS	Government Off-The-Shelf
HTML	Hypertext Markup Language
HTTPS	Hypertext Transfer Protocol Secure
ICT	Information and Communication Technology
IDA	Institute for Defense Analyses
IDE	Integrated Development Environment
IDS	Intrusion Detection System
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
IM	Instant Messaging
IPS	Intrusion Prevention System (IPS)
ISO	International Organization for Standardization
IT	Information Technology
JFAC	Joint Federated Assurance Center (DoD)
KPP	Key Performance Parameter
NIST	(U.S.) National Institute of Standards and Technology
NVD	(U.S.) National Vulnerability Database
OCI	Open Container Initiative
OSS	Open Source Software
OTS	Off-The-Shelf
O-TTPS	Open Trusted Technology Provider Standard
OVF	Open Virtualization Format
OWASP	Open Web Application Security Project
PKI	Public Key Infrastructure
PPP	Program Protection Plan
PyPI	Python Packaging Index
RFP	Request for Proposal
RA	Risk Assessment
RM	Risk Management
RMF	Risk Management Framework

SA	System and Services Acquisition
SCRM	Supply Chain Risk Management
SDE	Software Development Environment
SDK	Software Development (Tool)Kit
SEE	Software Engineering Environment
SFTP	SSH (or Secure) File Transfer Protocol
SS KPP	System Survivability Key Performance Parameter
SSH	Secure Shell
SOAR	State-of-the-Art Resources, see [Wheeler2016]
STIG	Security Technical Implementation Guide
SwA	Software Assurance
T&D	Test and Development
T&E	Test and Evaluation
TLS	Transport Layer Security
TSDM	Trusted Software Development Methodology
TSM	Trusted Software Methodology
TSN	Trusted Systems and Networks
UCS	Universal Coded Character Set
UTF-8	Unicode/UCS Transformation Format
VC	Version Control
VM	Virtual Machine
VMM	Virtual Machine Monitor (aka hypervisor)
VNC	Virtual Network Computing
VPN	Virtual Private Network
XML	Extensible Markup Language
XXE	XML External Entity

Bibliography

Note that URLs may have changed since publication. In some cases, we provide notes about the source material.

[Ahner 2017] Ahner, Darryl K and Bill Rowell. “T&E of Warfighting System Cyber Security Capabilities.” http://itea.org/images/pdf/conferences/2017_Cyber/Proceedings/AHNER%20ROWELL%20-%20ITEA_Cyber_Security_Workshop.pdf

[Andrews 2003] Andrews, Jeremy. 2003-11-05. “Linux: Kernel “Back Door” Attempt” KernelTrap. https://web.archive.org/web/20051122074510/https://kerneltrap.org/node/1584

[ArmyCoN] Army. Army Networthiness Program (Certificate of Networthiness). Retrieved 2017-11-29. https://www.atsc.army.mil/tadlp/implementation/config/networthiness.asp

[ASD STIG 2018] *Application Security and Development (ASD) Security Technical Implementation Guide (STIG), Ver 4, Rel 5.* 2018-01-26. https://iasecontent.disa.mil/stigs/zip/U_ASD_V4R5_STIG.zip

[Barrett 2017] Barrett, Brian. “Kaspersky, Russia, and the Antivirus Paradox.” *Wired.* 2017-11-10. https://www.wired.com/story/kaspersky-russia-antivirus/

[BBC 2017] BBC News. 2017-06-28. “Global ransomware attack causes turmoil.” BBC News. http://www.bbc.com/news/technology-40416611

[Biba1975] Biba, K.J. *Integrity Considerations for Secure Computer Systems*, MTR-3153, The MITRE Corporation, June 30, 1975. http://seclab.cs.ucdavis.edu/projects/history/papers/biba75.pdf

[Bell Labs 1979] Bell Labs. *Unix Seventh Edition Manual.* 1979. https://s3.amazonaws.com/plan9-bell-labs/7thEdMan/index.html

[Boyens2015] Boyens, Jon, Celia Paulsen, Rama Moorthy, Nadya Bartol. April 2015. *Supply Chain Risk Management Practices for Federal Information Systems and Organizations*. NIST Special Publication 800-161. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-161.pdf

[BSIMM2017] Building Security in Maturity Model (BSIMM). 2017. https://www.bsimm.com/

[Cimpanu 2017-CCleaner] Cimpanu, Catalin. 2017. “CCleaner Compromised to Distribute Malware for Almost a Month.” BleepingComputer. https://www.bleepingcomputer.com/news/security/ccleaner-compromised-to-distribute-malware-for-almost-a-month/

[Cimpanu 2017-ExpensiveWall] Cimpanu, Catalin. 2017-09-14. “Developers Unwittingly Embedded Malware in Their Android Apps via Shady SDK.” Bleeping Computer.

https://www.bleepingcomputer.com/news/security/developers-unwittingly-embedded-malware-in-their-android-apps-via-shady-sdk/

[Cimpanu 2017-JavaScript] Cimpanu, Catalin. 2017-08-04. “JavaScript Packages Caught Stealing Environment Variables.” Bleeping Computer. https://www.bleepingcomputer.com/news/security/javascript-packages-caught-stealing-environment-variables/

[CNSSI 4009] Committee on National Security Systems (CNSS) Instruction 4009. *National Information Assurance (IA) Glossary.* 2015-04-06. https://www.cnss.gov/CNSS/issuances/Instructions.cfm

[CNSSI 1253] Committee on National Security Systems Instruction 1253, *Security Categorization and Control Selection for National Security Systems,* March 15, 2012, March 27, 2014, as amended. https://www.cnss.gov/CNSS/issuances/Instructions.cfm

[CNSS 2017] Committee on National Security Systems (CNSS). 2017-07-26. *Supply Chain Risk Management*. Directive 505. https://www.cnss.gov/CNSS/issuances/Directives.cfm

[Cockerill 2015] Cockerill, Aaron. 2015-09-20. “Hundreds of millions of devices potentially affected by first major iOS malware outbreak.” Lookout Blog. https://blog.lookout.com/xcodeghost

[Coviello 2011] Coviello, Jr., Arthur W. (in his capacity as Executive Chairman, RSA, The Security Division of EMC). 2011-10-04. “Written Testimony: U.S. House of Representatives, Permanent Select Committee on Intelligence.” https://fas.org/irp/congress/2011_hr/100411coviello.pdf

[Dart1992] Dart, Susan A., Robert J. Ellison, Peter H. Feiler, A. Nico Habermann, and edited by Peter Fritzson. “Overview of Software Development Environments.” January 1992. University of California at Irvine. http://www.ics.uci.edu/~andre/ics228s2006/dartellisonfeilerhabermann.pdf https://www.researchgate.net/publication/237774668_Overview_of_Software_Development_Environments

[DISA Enclave 2016] Defense Information Systems Agency (DISA), 2016-01-22, *Enclave Test and Development STIG Overview,* V1R3 DISA http://iasecontent.disa.mil/stigs/zip/Jan2016/U_Enclave_Test_and_Development_V1R3_STIG.zip

[DoD PPP] DoD PPP Outline. July 2011. *Program Protection Plan Outline & Guidance,* Version 1.0. http://acqnotes.com/acqnote/careerfields/program-protection-plan

[DoD PPP Template] Protection Plan Template. http://acqnotes.com/acqnote/careerfields/program-protection-plan

[DoDI 5000.02] DoD Instruction (DoDI) 5000.02, *Operation of the Defense Acquisition System.”* 10 Aug 17 (Change 3). http://acqnotes.com/acqnote/acquisitions/dodi-5000

[DoDI 5200.44] DoD. July 27, 2017. *Protection of Mission Critical Functions to Achieve Trusted Systems and Networks (TSN).* DoD Instruction 5200.44. http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/520044p.pdf

[DoDI 8500.01] Department of Defense (DoD). 2014-03-14. *Cybersecurity.* DoDI 8500.01. http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/850001_2014.pdf

[DoDI 8510.01] DoD Instruction 8510.01, *Risk Management Framework (RMF) for DoD Information Technology (IT).* July 28, 2017. http://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/851001_2014.pdf

[EMC 2011] EMC, RSA Division. 2011-07. Frequently asked questions about RSA secured: Information for RSA Customers. Customer faq 0711. https://www.emc.com/collateral/guide/11455-customer-faq.pdf

[Fruhlinger 2017] Fruhlinger, Josh. 2017-10-17. "Petya ransomware and NotPetya malware: What you need to know now." CSO Online [Not confirmed to be an abbreviation]. https://www.csoonline.com/article/3233210/ransomware/petya-ransomware-and-notpetya-malware-what-you-need-to-know-now.html

[GE 1991] General Electric (GE). 1991-12-23. *Trusted Software Development Methodology (TSDM) Report. Volumes 1 and 2.* Product numbers A075-101A and A075-102A. For the Strategic Defense Initiative (SDI) Organization, later known as the Missile Defense Agency. A minor update was later published on 1993-07-02.

[Goodin 2015] Goodin, Dan. 2015-09-21. "Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store: Outbreak may have caused hundreds of millions of people to download malicious apps." *Ars Technica*. https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/

[Goodin 2017PyPI] Goodin, Dan. 2017-09-16. "Devs unknowingly use "malicious" modules snuck into official Python repository: Code packages available in PyPI contained modified installation scripts." *Ars Technica*. https://arstechnica.com/information-technology/2017/09/devs-unknowingly-use-malicious-modules-put-into-official-python-repository/

[Greenberg 2017] Greenberg, Andy. 2017-09-18. "Software has a Serious Supply-Chain Security Problem." *Wired*. https://www.wired.com/story/ccleaner-malware-supply-chain-software-security/

[Greenberg 2017-Petya] Greenberg, Andy. 2017-07-07. "The Petya Plague Exposes the Threat of Evil Software Updates." *Wired.* https://www.wired.com/story/petya-plague-automatic-software-updates/

[Goodin 2017-GitHub] Goodin, Dan. 2017-03-29. "Someone is putting lots of work into hacking GitHub developers: Dimnie recon trojan has flown under the radar for three years... until now." *Ars Technica*. https://arstechnica.com/information-technology/2017/03/someone-is-putting-lots-of-work-into-hacking-github-developers/

[Hern 2014] Hern, Alex. 2014-10-28. "Tor users advised to check their computers for malware: Users of the anonymising service may have accidentally downloaded malware thanks to a malicious Russian hacker." *The Guardian.* https://www.theguardian.com/technology/2014/oct/28/tor-users-advised-check-computers-malware

[Hussein 2017] Hussein, Mahmoud, Reda Nouacer, and Ansgar Radermacher. 2017-09-28. “Towards a Safe Software Development Environment.” IEEE Xplore. http://ieeexplore.ieee.org/document/8049827/

[JFAC 2015] Joint Federated Assurance Center (JFAC). 2015. Joint Federated Assurance Center (JFAC) Charter. Signed by Robert O. Work (Deputy Secretary of Defense) on 2015-02-09. https://www.acq.osd.mil/se/docs/JFAC-Charter-020915-SansMemo.pdf

[Kaspersky 2017] Kaspersky Labs. 2017-08-15. “ShadowPad: How Attackers hide Backdoor in Software used by Hundreds of Large Companies around the World: ShadowPad is one of the largest known supply-chain attacks. Had it not been detected and patched so quickly, it could potentially have targeted hundreds of organizations worldwide.” https://www.kaspersky.com/about/press-releases/2017_shadowpad-how-attackers-hide-backdoor-in-software-used-by-hundreds-of-large-companies-around-the-world

[Kupsch 2016] Kupsch, James A., Elisa Heymann, Barton Miller, and Vamshi Basupalli. 2016-04-29. “Bad and good news about using software assurance tools.” Later published in *Software Practice and Experience* (after online publication), Volume 47, Issue 1, January 2017, pp. 143–156. http://onlinelibrary.wiley.com/doi/10.1002/spe.2401/full

[LF 2017] CII Best Practices Badge Program. https://bestpractices.coreinfrastructure.org/

[Maunder 2017] Maunder, Mark. 2017-08-17. “PSA: 4.8 Million Affected by Chrome Extension Attacks Targeting Site Owners”. Wordfence. https://www.wordfence.com/blog/2017/08/chrome-browser-extension-attacks/

[Mueller 2012] Mueller, Carl J. August 27, 2017. “Securing the Software Development Environment.” TAMU-Central Texas in Killeen, Infosec Institute. http://resources.infosecinstitute.com/securing-the-software-development-environment/. This has some old references, e.g., CVS & Peek, but it does note the risk of malicious developers.

[NICCS] National Initiative for Cybersecurity Careers and Studies (NICCS). NICCS Glossary. https://niccs.us-cert.gov/glossary

[NIST SP 800-37] NIST Special Publication (SP) 800-37 revision 1. *Guide for Applying the Risk Management Framework to Federal Information Systems: a Security Life Cycle Approach.* 2014-06-10. https://csrc.nist.gov/publications/detail/sp/800-37/rev-1/final

[NIST SP 800-39] NIST Special Publication (SP) 800-39. *Managing Information Security Risk.* 2011-03. http://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-39.pdf

[NIST SP 800-53] NIST Special Publication (SP) 800-53 rev 4. *Security and Privacy Controls for Federal Information Systems and Organizations.* Includes updates to 2015-01-22. https://csrc.nist.gov/publications/detail/sp/800-53/rev-4/final

[OpenGroup 2014] Open Group. 2014-07-29. *Open Trusted Technology Provider Standard (O-TTPS), Version 1.1.* Identical to ISO/IEC 20243:2015. https://publications.opengroup.org/c147

[OWASP 2008] Open Web Application Security Project (OWASP). 2008-06-20. Threat agent template. https://www.owasp.org/index.php/Threat_agent_template Retrieved 2017-12-27.

[OWASP 2017] Open Web Application Security Project (OWASP). 2017. Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project

[Popek 1974] Popek, Gerald J., and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures." Communications of the Association for Computing Machinery (ACM) (CACM). Volume 17 Issue 7, July 1974. Pages 412-421. https://dl.acm.org/citation.cfm?doid=361011.361073

[Reddy 2014a] Reddy, Dan. March/April 2014. Collaborating Across the Supply Chain to Address Taint and Counterfeit. *CrossTalk.* https://www.crosstalkonline.org/storage/issue-archives/2014/201403/201403-Reddy.pdf

[Reddy 2014b] Reddy, Dan. July 2014. Criticality analysis and the supply chain: Leveraging representational assurance. Technovation. http://dx.doi.org/10.1016/j.technovation.2014.01.009 and http://www.sciencedirect.com/science/article/pii/S0166497214000108

[Reproducible] Reproducible builds. Definitions. https://reproducible-builds.org/docs/definition/

[Rowell] Rowell, William F. and Darryl K. Ahner. "Improving the T&E Workforce's Understanding of the New Approach to Developing Warfighting System Cyber Requirements." The ITEA Journal of Test and Evaluation. September 2017.

[Saltzer&Schroeder 1975] Jerome H. Saltzer, and Michael D. Schroeder. "The Protection of Information in Computer Systems." Invited tutorial paper. *Proceedings of the IEEE* 63, 9 (September 1975). pp. 1278–1308. Reprinted in Protection of Information in Computer Systems. IEEE 1975 CompCon tutorial, David D. Clark and David D. Redell, editors, IEEE # 75CH1050-4. Also reprinted in *Advances in Computer System Security,* Rein Turn, editor, ArTech House, Dedham, MA, 1981, pages 105–135, ISBN 0-89006-096-7. Also reprinted in "Protecting Data & Information: A Workshop in Computer & Data Security," by Marvin S. Levin, Steven B. Lipner, and Paul A. Karger, Digital Equipment Corporation, 1982. A draft dated October 3, 1974, was circulated locally as M.I.T. Project MAC Computer Systems Research Request for Comments #60. http://web.mit.edu/Saltzer/www/publications/protection/index.html http://www.cs.virginia.edu/~evans/cs551/saltzer/

[Shaw 2017a] Shaw, Richard A. October 30, 2017. "Software Supply Chain Attacks." https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Placemat.pdf

[Shaw 2017b] Shaw, Richard A. October 30, 2017. "Software Supply Chain Attacks": Reference List. https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Reference%20List.pdf

[Simpson 2010] Simpson, Stacy (editor). 2010-06-14. *Software Integrity Controls: An Assurance-Based Approach to Minimizing Risks in the Software Supply Chain.* SAFECode. June 14, 2010.

http://www.safecode.org/publication/SAFECode_Software_Integrity_Controls0610.pdf

[Simpson 2011] Simpson, Stacy (editor). 2011-02-08. *Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today. 2nd Edition.* https://www.safecode.org/publication/SAFECode_Dev_Practices0211.pdf. Note that there is a third edition, [SAFECode 2018], but that version does not discuss sandboxing.

[SAFECode 2018]. 2018-03. *Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Lifecycle Program, 3rd edition.* SAFECode. https://www.safecode.org/publications/

[Sutherland 1989] Sutherland, D. 1989-01. "Designing the software development environment: a case study." *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track.* http://ieeexplore.ieee.org/document/48113/#full-text-section

[Tschacher 2016] Tschacher, Nikolai Philipp. 2016. *Typosquatting in Programming Language Package Managers.* Thesis. University of Hamburg Department of Informatics. http://incolumitas.com/data/thesis.pdf

[Wagoner] Wagoner, Larry. Date Unstated. "Software Assurance Tool Status and Gaps - System Assurance PTF." National Security Agency. http://sysa.omg.org/docs/swa_nsa.pdf

[Wheeler 2010] Wheeler, David A. (leader of Key Practices Group) et al. for the Supply Chain Risk Management (SCRM) Program Office, Trusted Mission Systems and Networks Directorate. 2010-02-25. "Key Practices and Implementation Guide for the DoD Comprehensive National Cybersecurity Initiative 11 – Supply Chain Risk Management Pilot Program." https://rmfks.osd.mil/rmf/Guidance/RMFRelatedTopics/Pages/SCRM.aspx

[Wheeler 2015-scm] Wheeler, David A., 2015, "Software Configuration Management (SCM) Security." https://www.dwheeler.com/essays/scm-security.html

[Wheeler 2015-programming] Wheeler, David A. 2015-09-19. "Secure Programming HOWTO." https://www.dwheeler.com/secure-programs/

[Wheeler&Reddy 2015] Wheeler, David A. and Dan Reddy. 2015. "Countering Development Environment Attacks," RSA 2015, http://www.rsaconference.com/events/us15/agenda/sessions/1613/countering-development-environment-attacks

[Wheeler 2016] Wheeler, David A., and Amy E. Henninger. November 2016. *State-of-the-Art Resources (SOAR) for Software Vulnerability Detection, Test, and Evaluation* 2016 (aka SOAR Report). https://www.acq.osd.mil/se/docs/P-8005-SOAR-2016.pdf or https://www.acq.osd.mil/se/initiatives/init_jfac.html

[Wheeler 2017cloud] Wheeler, David A. 2017-04-19. "Cloud Security: Virtualization, Containers, and Related Issues." https://www.dwheeler.com/essays/cloud-security-virtualization-containers.html

[Wheeler 2017Heartbleed] Wheeler, David A. 2017-01-29. How to Prevent the next Heartbleed. https://www.dwheeler.com/essays/heartbleed.html

[Woiciechowski 2013] Woiciechowski, Stephanie (EMC Product Security Office, EMC Corporation), 2013, “Source Code Protection: Evaluating Source Code Security.” https://www.slideshare.net/perforce/whitepaper-source-code-protection

[Wright 2014] Wright, Madeline Wright and Dr. Carl Mueller. Nov/Dec 2014. “Soft Locking Down the Software Development Environment.” *Crosstalk*. http://www.crosstalkonline.org/storage/issue-archives/2014/201411/201411-Wright.pdf

[Wright 2017] Wright, Madeline and Carl Mueller. 2017. “Locking Down the Software Development Environment.” *Crosstalk.* http://m.crosstalkonline.org/issues/16/142/ http://cross5talk2.squarespace.com/storage/issue-archives/2017/201707/201707-0-Issue.pdf http://static1.1.sqspcdn.com/static/f/702523/25771604/1418603717260/201411-0-Issue.pdf?token=EaiuLIiDQo57KcMgBFU3VqZsqpU%3D

[Xiao 2015a] Xiao, Claud. 2015-09-17. “Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store.” Palo Alto Networks. https://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/

[Xiao 2015b] Xiao, Claud. 2015-09-18. “Malware XcodeGhost Infects 39 iOS Apps, Including WeChat, Affecting Hundreds of Millions of Users.” Palo Alto Networks. https://researchcenter.paloaltonetworks.com/2015/09/malware-xcodeghost-infects-39-ios-apps-including-wechat-affecting-hundreds-of-millions-of-users/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (DD-MM-YY)	2. REPORT TYPE	3. DATES COVERED (From – To)
00-07-18	Final	

4. TITLE AND SUBTITLE

Securely Using Software Assurance (SwA) Tools in the Software Development Environment

5a. CONTRACT NUMBER

HQ0034-14-D-0001

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBERS

6. AUTHOR(S)

David A. Wheeler, Daniel J. Reddy

5d. PROJECT NUMBER

AU-5-3856

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311-1882

8. PERFORMING ORGANIZATION REPORT NUMBER

P-9166

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Thomas D. Hurt
Dir. JFAC, Dep.Dir. SW Eng./SW Assurance, OUSD(R&E) Enterprise Engineering
Engineering Enterprise
4800 Mark Center Dr., Suite 16D-08
Alexandria, VA 22350-3600

10. SPONSOR'S / MONITOR'S ACRONYM

OUSD(R&E)

11. SPONSOR'S / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

Project Leader: E. Kenneth Hong Fong

14.ABSTRACT Software assurance (SwA) may be defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner." Since modern systems are under constant attack, sufficient SwA is vital. In practice, a suite of SwA tools is necessary to help achieve this. However, there are potential challenges to securely using a suite of SwA tools. Software development environments (SDEs) are increasingly under focused attack, since subverting software during development can be easier than subverting it after it is deployed. One mechanism for subverting SDEs is to exploit vulnerabilities in its tools or to provide maliciously subverted tools to an SDE. The goal of this paper is to help ease the deployment of SwA tools, by countering potential objections to using them. To achieve this, we discuss how to protect against potential supply chain risks of SwA tools themselves, including how to protect the SDE in general against supply chain risks and how the mechanisms to counter SwA tool risks fit into the SDE. We show that it is possible to modify SDE practices to use a wide variety of SwA tools and still manage the inherent risks. Isolation mechanisms can be used, for example, to separate tools and restrict access for specific tasks. This approach can be automated and may reduce risk in a relatively uncomplicated manner. In particular, the "medium protection" approach discussed here should be easy to incorporate in existing SDEs. We recommend that organizations fully embrace the use of many SwA tools when developing software. Where appropriate, they should consider taking the additional steps discussed here if they determine that the risks of using SwA tools are otherwise too high. Our hope is that this information will lead to the widespread safe use of suites of SwA tools.

15. SUBJECT TERMS Software assurance; SwA; tool; SwA tool; software development environment; SDE; development environment; malicious code; malicious software; malicious tool; subversion; subverted tool; isolation; virtual machine; container; containerization; protection; medium protection; risk management; countermeasure; life cycle; supply chain; supply chain risk management; SCRM; computing environment; trust; trustworthy; trustworthiness; approved products list; certificate of networthiness; software engineering environment; integrated development environment; programming environment; static analysis; dynamic analysis; malicious supplier; insider threat; Joint Federated Assurance Center; JFAC; confidentiality; integrity; source code

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas D. Hurt
Unclassified	Unclassified	Unclassified	Unlimited	92	19b. TELEPHONE NUMBER (Include Area Code) 571-372-6129