Secure Software Development Across the Lifecycle for Space Systems

By: Brandon Bailey, Nick. Cohen, Evan Glick, Gregory Mulert, Henry Reed, Steven Rosemergy

White Paper written for Consultative Committee for Space Data Systems (CCSDS) Security Working Group – to be derived into Magenta Book for CCSDS

Approved for public release. OTR 2022-00085.

Contents

1 Purpose and Scope 1

2 Intended Audience/Applicability 1

3 Why Perform Secure Software Development 2

4 Architecture Methods to promote insight into security 3
- 4.1 Overview 3
- 4.2 Architecture Requirements: Quality Attributes 4
- 4.3 Specifying Quality Attribute Requirements 4
- 4.4 Architecture Tactics 6
  - 4.4.1 Architecture Tactics: Promoting Security and Reliability 6
  - 4.4.2 Tactics to Promote Availability 8
  - 4.4.3 Tactics to Promote Modifiability: Limiting Security Issue Ripple Effects 9

5 Secure Coding Requirements 10
- 5.1 DevSecOps Processes 12
  - 5.1.1 Fundamental Goals 12
  - 5.1.2 Code Repositories & CI/CD Pipelines 13
- 5.2 Secure Coding Guidelines 15
  - 5.2.1 Improper Input Validation (CWE-20) 17
  - 5.2.2 Permissions, Privileges, and Access Control (CWE-264) 18
  - 5.2.3 Data Processing Errors (CWE-19) 20
  - 5.2.4 Numeric Errors (CWE-189) 22
  - 5.2.5 Improper Restriction of Operations Within the Bounds of a Memory Buffer (CWE-119) 25
  - 5.2.6 Logic Errors (CWE-840) 27
  - 5.2.7 Pathname Traversal Errors (CWE-21) 29
  - 5.2.8 Insufficient Verification of Data Authenticity (CWE-345) 30
  - 5.2.9 Time and State (CWE-361) 31
  - 5.2.10 Bad Coding Practices (CWE-1006) 32
  - 5.2.11 Resource Management (CWE-399) 33
  - 5.2.12 Web Problems (CWE-442) 35
- 5.3 Commercial, Off-the-Shelf; Free and Open-Source Software; and Re-use Security Concerns 37
- 5.4 Software Bill of Materials 38
- 5.5 Compilation Guidelines 40
- 5.6 Authentication and Password Management 41

6 Analysis Tools and Techniques 42
- 6.1 Static Analysis 42
  - 6.1.1 Tool Choice 43
  - 6.1.2 Scanning 44
  - 6.1.3 Third Party Code Analysis 45
  - 6.1.4 Analysis Without Source Code 46
- 6.2 Dynamic Testing 48
  - 6.2.1 Preparing for Dynamic Testing 49
  - 6.2.2 Fuzzing Tools and Frameworks 51
  - 6.2.3 Symbolic and Concolic Execution 51
  - 6.2.4 Memory and Thread Sanitization 52
  - 6.2.5 Analyzing and Mitigating Findings 54

6.2.6 Testing Environments 55
6.2.7 Targeted Penetration Testing 55
7 Software Acquisition Security Best Practices 57
8 Software Security Authorization and/or Certification 58
9 Maintenance and Sustainment 58
9.1 Vulnerability Assessments 59
9.1.1 Vulnerability Assessment Requirements 59
9.2 COTS and FOSS 59
Appendix A: High Priority Common Weakness Enumerations 62
Appendix B: Secure Coding Design Review Checklist 75
Appendix C: Secure Coding Code Review Checklist 77
Appendix D: Sample Requirements 1
Appendix E: Glossary of Terms 1
Bibliography 5

Figure 1: Component of a Space System 1
Figure 2: Bake Security in Early...2
Figure 3: Open-Source Vulnerability by Language (2009-2018) (WhiteSource, n.d.) 3
Figure 4: Software Security Across the Lifecycle 3
Figure 5: Quality Attribute Parts..4
Figure 6: Summary Security Tactics (Bass & Clements, 2003) 6
Figure 7: Summary Availability Response Tactics (Bass & Clements, 2003) 8
Figure 8: NIST Controls Applicable to Secure SW Development 12
Figure 9: XSS Attack 36
Figure 10: Nutritional Facts Example..38
Figure 11: NTIA Baseline SBOM 39
Figure 12: Microsoft Published SPDX Fields...39
Figure 13: SBOM Use Cases 40
Figure 14: Example Testing Process..49
Figure 15: Analyzing Faults from Testing...54
Figure 16: Example Penetration Testing Process 56

1 PURPOSE AND SCOPE

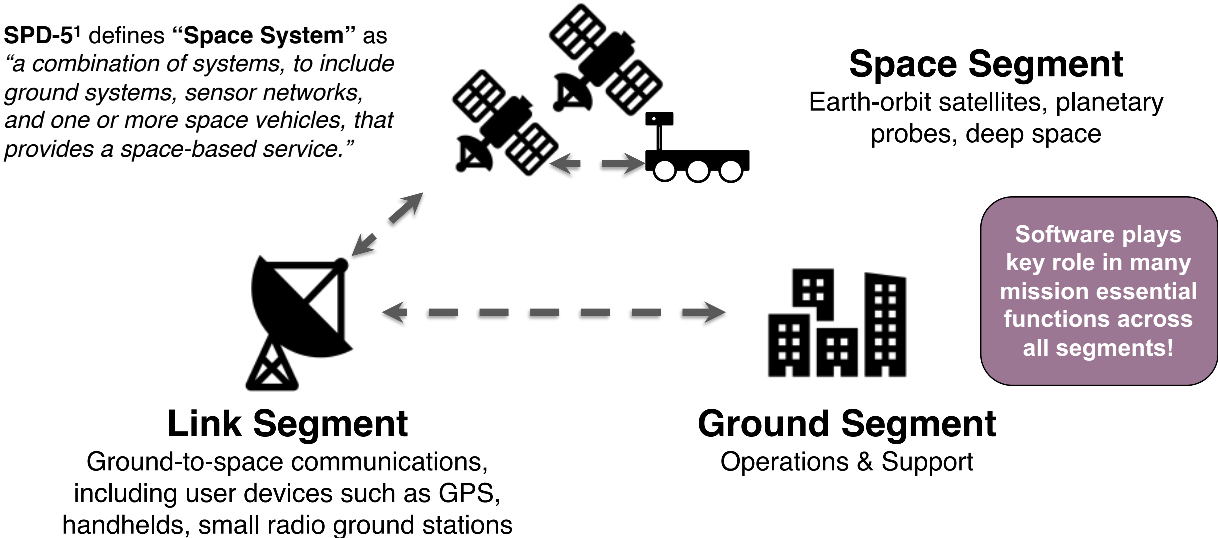
The purpose and scope of this document is to provide guidance to CCSDS missions for implementing secure software development throughout the Software Design Life Cycle. The scope focuses on space systems and spans the acquisition lifecycle from early requirements definition to sustainment and maintenance. This document is comprised of industry best practices, subject matter expertise based on experience with space programs. It contains procedures, processes, guidelines, and design strategies for developing secure code, systems, and projects. This report includes recommendations for practices mission owners should adopt.

2 INTENDED AUDIENCE/APPLICABILITY

The recommended audience is software developers, software project managers, and software assurance (SwA) professionals. The authors assume knowledge and experience writing space mission software and knowledge of software design patterns, general software design principles, resource management such as memory, files, and threats, and development processes such as Development Operations (DevOps). One purpose of this document is to familiarize the audience with security issues that arise in software development; therefore, the audience does not need extensive experience with software security but knowledge of basic security issues that arise in software systems such as buffer overflows would be helpful.

The applicability is to ground system software developers as well as flight software developers as software plays a key role in mission essential functions across all segments as depicted below.

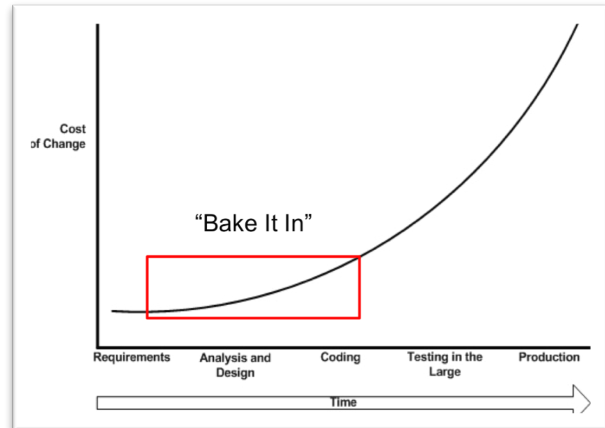
FIGURE 1: COMPONENT OF A SPACE SYSTEM



3 Why Perform Secure Software Development

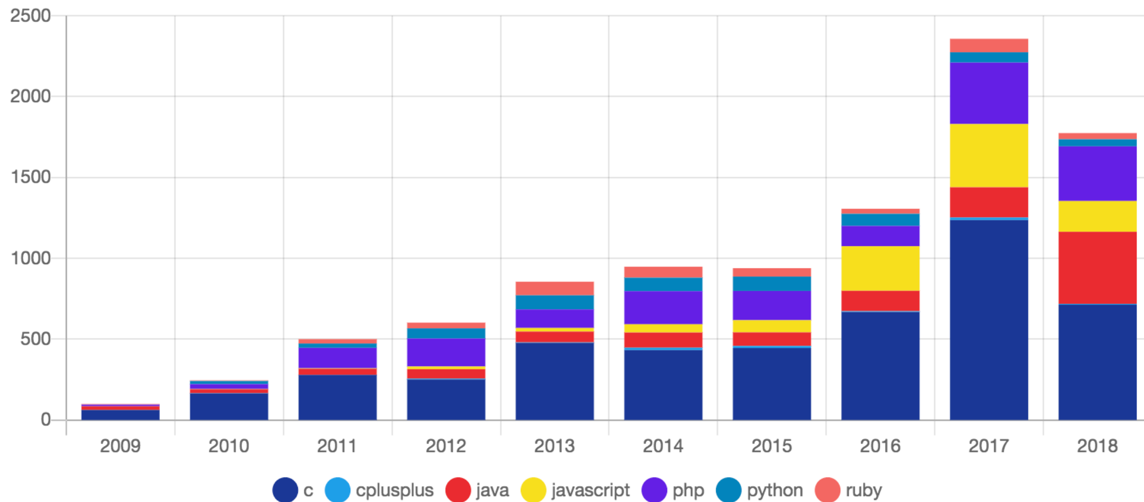
Software is a significant risk that requires oversight, investment, secure software development practices and assurance by developers and/or third-party entities. In traditional systems, it is estimated that approximately 80% of system functionality is or will soon be provided through software. Space systems are no different as the lines of code performing mission critical functionality has expanded exponentially over the past 30 years. In traditional systems, 80-90% of breaches come from weaknesses in the software with about half of the software weaknesses being attributed to architectural security flaws. Security issues are often too difficult to find with the human eye, especially in the architecture. Nearly 90% of software flaws are introduced in software design & development which requires action from management and development organizations alike. Investment is needed but there is a return on the software security investment. 60% of software flaws are not found until later in integration and system test requiring up to 25 hours to fix per issue vs 15 minutes during development. There is a need to shift the discovery of any defect, including security to the left to prevent mission impact and cost to mitigate.

Figure 2: Bake Security in Early



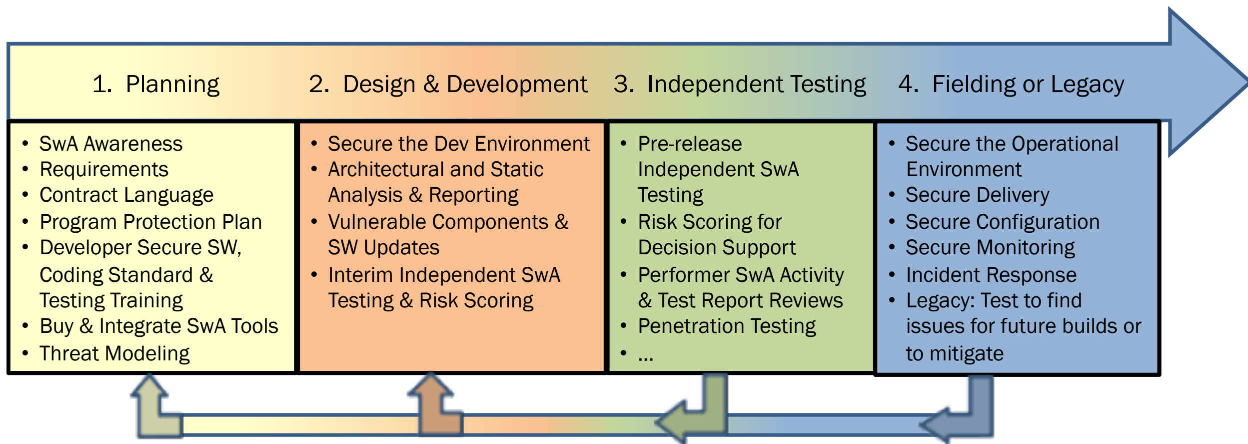
An example of why secure software development principles is extremely important can be explained when discussing the flight software for a mission. An integral component of the Command and Data Handling (C&DH) system development is the flight software (FSW), usually written in C/C++. It is often structured as a state machine, controlling the transitioning of the spacecraft between its operational modes that define its actions and behavior. C/C++ is widely used in the development of software for embedded systems, even mission-critical and hard-real-time systems. C++ (as well as C), by its design, may not be as suitable language as once thought for writing high integrity or mission-critical software. C++ gives a programmer a great deal of freedom. With freedom comes responsibility, though, and, in the case of C++, a whole lot of responsibility. Nevertheless, when used properly there are good reasons for using C++ in a mission-critical system like a spacecraft. When using C++ for mission-critical systems, attention must be paid to avoid any language constructs and code that can potentially lead to unintended program behavior. Coding standards that limit language features to a safe subset that can be used are critical as well as including tools for automatic validation of these standards. (Obiltschnig, n.d.) Open-source software vulnerabilities can be used as an indicator on the prevalence of vulnerabilities within software which indicates C/C++ has the most vulnerabilities per language in the past 10 year. (WhiteSource, n.d.) The point of this metric is to communicate that FSW is typically written in languages that have a history of vulnerabilities specifically with input sanitization and buffer conditions (i.e., CWE-20, CWE-119).

FIGURE 3: OPEN-SOURCE VULNERABILITY BY LANGUAGE (2009-2018) (WHITESOURCE, N.D.)



The goal is to start as early as possible with secure software development and any third-party SwA activities. Start small, start early & leverage security minded developers and SwA experts across the lifecycle. Various aspects of below will be described in subsequent sections.

FIGURE 4: SOFTWARE SECURITY ACROSS THE LIFECYCLE



4 ARCHITECTURE METHODS TO PROMOTE INSIGHT INTO SECURITY

4.1 OVERVIEW

Because Software Architecture principles and practice is a broad topic, the discussion in this section to architectural principles with an emphasis on promoting reliability and security. It touches on architectural practices that enable shared understanding of architectural decisions and the identification of risks. For more information on these and other architectural principles and practices see, *Software Architecture in Practice* (Bass & Clements, 2003).

4.2 Architecture Requirements: Quality Attributes

When considering architectural decision making, it requires visionary future proofing especially when architecting for threats and vulnerabilities that are yet to be known. The visionary architect will need to anticipate and balance the long-term needs, while simultaneously providing short-term guidance that enables the production of working software. The truth of the matter is that this visionary architect's job is made possible based on the understanding that the functionality of any given system and its quality attributes are orthogonal to each other.

While this statement may run counter to what we believe about architectural decision making; consider how one might allocate functionality to optimize performance or how the system should respond to faults, failures, or malicious insiders? Often these questions go unanswered until problems arise (i.e., operating system patch that exposes a race condition, user error that leads to mission failure, etc.).

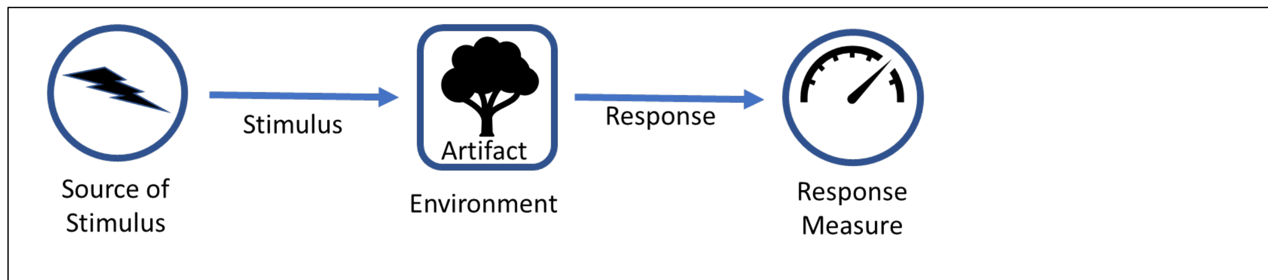
This where quality attributes come into play. For any given system to maintain utility it is designed to balance systemic attributes, or quality attributes, based on their relative priority to all stakeholders – it is not enough to design a system to promote code modifiability (the first instinct of any developer) without balancing security, performance, testability, or usability. For any given system, the balance of quality attributes exists in tension with each other – when portability is highly optimized (e.g., software language, compiler, operating system, database, and hardware specific dependencies are isolated), portability can be achieved at the cost of performance. When the architect balances these two quality attribute concerns (e.g., address risk of database vendor lock by isolating vendor specific functions in database functions), performance and portability goals can be achieved.

4.3 Specifying Quality Attribute Requirements

Achieving a balance of quality attributes first requires that the primary driving quality attributes be identified, prioritized, and fully characterized. Once characterized, the architect can select the specific architectural principles that best promote and inhibit the desired architectural qualities. The most effective method for specifying quality attributes, suitable for both architectural reasoning and technical driver prioritization is through the use of *Quality Attribute Scenario Descriptions* (Bass & Clements, 2003).

Figure 5: Quality Attribute Parts

Quality attribute scenario description represents a quality attribute requirement and is made up into six parts. They are:



1. Source of Stimulus: An entity that generates a stimulus. This may be user of the system, other systems or subsystem that are either internal or external to this system, time, or hardware (e.g., actuator or sensor).
2. Stimulus: The condition, trigger, or event that occurs and is under consideration
3. Environment: The conditions under which the stimulus can occur. This may include quiescent conditions, heavy load, start-up, shutdown
4. Artifact: The part of the system that is stimulated by the stimulus.
5. Response: The expected activity assumed after the stimulus arrives.
6. Response Measure: The quantifiable performance of the response.

Extending this concept into the security domain, consider the insider threat attack scenario where the source stimulus is an authorized user of the software. This is a good example where quality and security overlap. Some may call this quality attribute exercise threat modeling which is a very common practice during early stages of architectural design.

Example #1: Insider Threat Scenario. Consider an attack scenario whereby a correctly identified and authorized user of a given system modifies mission initialization data. In this case, the expected response of the system is to be able roll-back configuration settings for up to two days.

For this scenario, the quality attribute parts are:

TABLE 1 ATTRIBUTE SCENARIO DESCRIPTION EXAMPLE 1

Quality Attribute Step	Description
Source of Stimulus	Authorized User
Stimulus	Modifies system configuration
Artifact	System Datastore or database
Environment	Steady State (normal Ops)
Response	System records state prior to changes and can restore previous configuration settings
Response Measure	System can restore previous state for up to 48 hours.

Example #2: System Availability Scenario. Consider system availability scenario whereby one or more parts of the system lose network connectivity or has a loss of power. In this case, the expected response of the system is to notify operational users, of outage, timeline for outage notification is 6 seconds.

For this scenario, the quality attribute parts are:

TABLE 2 ATTRIBUTE SCENARIO DESCRIPTION EXAMPLE 2

Quality Attribute Step	Description
Source of Stimulus	Network Switch
Stimulus	Loss of network traffic
Artifact	Networked subsystems
Environment	Steady State (normal Ops)
Response	User Interface provides a visible indicator of outage and outage information.
Response Measure	Notification occurs within 6 seconds of detected outage.

Note that in both scenario description examples, neither specify any specific technical solutions (e.g., rollback, or heartbeat) in response to the stimuli, rather, what the expected response of the system should be. As simple as these scenarios are, oftentimes quality attribute requirements related to security and availability are oftentimes unarticulated, or wrongly assumed to be captured under "non-functional" requirements. Articulating quality attribute scenarios into the six-part descriptions enable the architect, developers, and involved business stakeholders to characterize the appropriate response and response measures to enable the prioritization of quality attributes relative to each other – and the selection of the most appropriate architectural tactics to employ.

4.4 ARCHITECTURE TACTICS

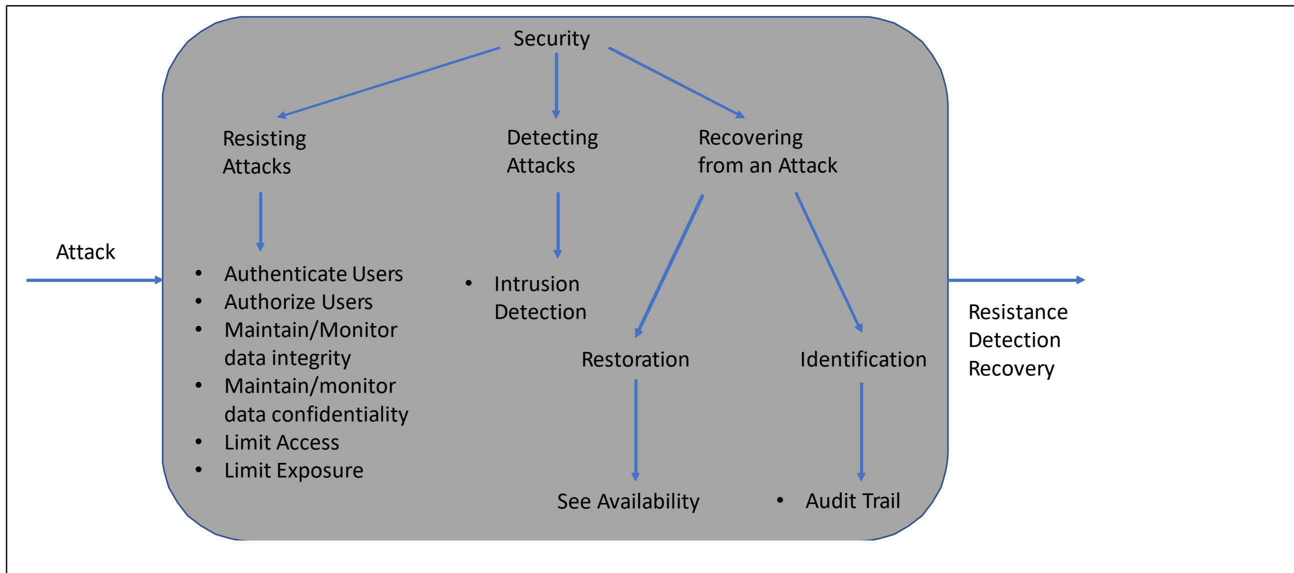
Architecture represents the individual pieces in a large puzzle, when combined with the allocation of responsibility (functional allocation) they make up an overall software architecture. The total collection of applied tactics is often referred to as the *architecture strategy*, as it represents the explicit design choices that are traced to specific quality attribute scenarios (Bass & Clements, 2003). When we consider each quality attribute scenario individually, applying one or more architecture tactics may be necessary in order to achieve the desired (or acceptable) response in light of competing architectural drivers.

4.4.1 ARCHITECTURE TACTICS: PROMOTING SECURITY AND RELIABILITY

WHEN WE CONSIDER SECURITY TACTICS (SHOWN IN

Figure 6), they are in response to attacks. The often-expected response to an attack is to resist but it may be more appropriate to detect and recover in some cases.

FIGURE 6: SUMMARY SECURITY TACTICS (BASS & CLEMENTS, 2003)



4.4.1.1 Tactics for Detecting Attacks

The classic "man-in-the-middle" class of attacks involve insertion of a third party into a communications channel, allowing messages to be intercepted, captured, and/or manipulated before forwarding them on to their destination. These kinds of attacks prey upon the assumption that once a secure, encrypted connection has been established between two endpoints, communication over that channel can be implicitly trusted. Tactics to detect these types of attacks include *Data Integrity Detection* and *Anomaly Detection.*

Data Integrity Detection. Verification of message integrity and detection of message delay are two methods of *detecting* these types of attacks. Verification consists of comparing the data that were initially transmitted with the data that are received. At the implementation level this is done using cryptographic signatures. The originator signs the payload of the original message, and that signature is verified by the receiver. If the signature is not valid based on the sender's public key, it is an indication that the data was somehow tampered with in transit.

Anomaly Detection. In cases where there is compromised hardware or software, the persistent threat may be exfiltration of credentials and/or data. In such cases, tactics to characterize nominal and off-nominal payload sizes and frequency may be used to trigger notifications.

4.4.1.2 Tactics to promote Attack Resistance

The common denominator in many attacks is a form of privilege escalation, in which the attacker gains access to data or other resources without appropriate authorization. This can range from reading protected rows in a database to sending unauthorized commands to Space Vehicle (SV) thrusters, or worse. While no software component can be made completely impenetrable to attack, especially in cases where hardware is compromised, the tactic of "least privilege" can mitigate the impact of attacks. Least privilege combines three architectural tactics: ***Authenticate Users***, ***Authorize Users****, and* ***Limit Access*** to any given system. Each of these tactics are separate from each other and when employed together and combined with ***Data Integrity Detection*** they establish an integrated attack resistance to attacks, capable of detecting and resisting attacks from compromised components within the system.

When these tactics are combined, such that software functions are only granted access to the resources required to accomplish its task, access to critical system components is compartmentalized. This makes the attacker's task significantly more difficult than dealing with a target that has been hardened using orthogonal resistance tactics.
Keeping in mind that some resistance tactics may incur additional overhead that may limit system utility, careful selection and application of combined tactics are recommended to assure a balance of concerns.

4.4.1.3 TACTICS TO PROMOTE ATTACK RECOVERY

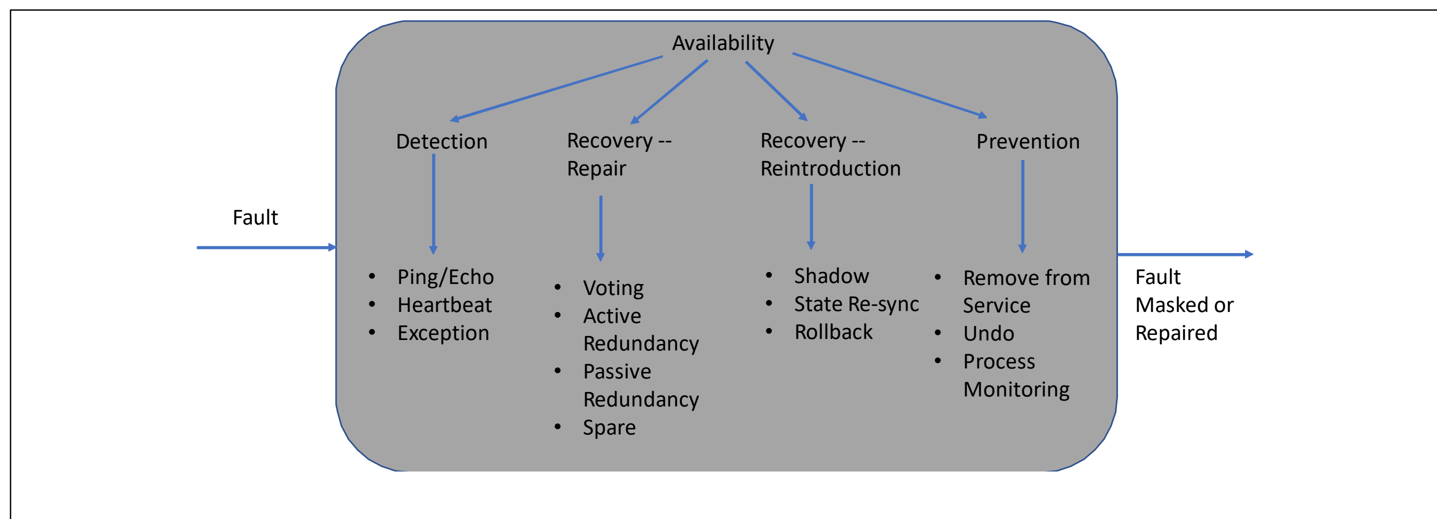
In the aftermath of an attack that involves compromised hardware or software, there is a lingering question: How can we ensure the attack vector has been eliminated? In many cases, replacing the hardware with a known good alternative is simply not an option. Replacing software may be more feasible but is still likely to require significant time and effort. Until it can be deployed, the vulnerability remains.

An architectural tactic that can address this concern is ***voting***. Voting requires multiple separate instances (and sometimes independent implementations) of the software component to operate in parallel. In the ideal case, these instances would run on heterogeneous hardware. Before carrying out an action that could impact the system's ability to perform its core functions, all of the instances must be polled and reach a consensus. If one of the instances is compromised, it will be "outvoted" by the non-compromised instances, preventing the malicious behavior from occurring.

4.4.2 TACTICS TO PROMOTE AVAILABILITY

When we consider availability tactics (shown in Figure 7), they are in response to one or more faults in the system. The often-expected response to a fault may be to recover and repair, but it may also be to detect and report faults. When considering a particular architectural tactic to promote availability, one must first start with a fault, then determine which explicit response tactics most appropriately promote availability.

FIGURE 7: SUMMARY AVAILABILITY RESPONSE TACTICS (BASS & CLEMENTS, 2003)



4.4.2.1 Tactics to promote Fault Prevention

The stereotypical image of an attacker is someone who infiltrates and subverts a system to serve their own ends. However, it is much simpler for an attacker to merely disrupt a system's operation, making it unable to carry out its mission. This can be accomplished by overloading any limited resource, including memory, processing power, network I/O, or storage. To guard against these types of attacks, an architecture can be designed to identify them and pre-emptively remove the targeted component from service in a controlled manner. This kind of proactive response leaves the defender in control, rather than the attacker. For example, if an attacker attempts to overload a system by sending a huge volume of counterfeit requests, the communication channel can be closed or throttled until the attack subsides. An attempt to exploit memory leaks can be mitigated by rebooting the system before out-of-memory errors occur.

4.4.3 Tactics to Promote Modifiability: Limiting Security Issue Ripple Effects

When we consider modifiability tactics (i.e., compromised software or hardware components), they are in response to changes that need to occur, whether they be in the implementation or while the system is running. The often default response to a change is to modify the underlying source code and rebuild/redeploy the system software. When considering a particular architectural tactic to promote modifiability (i.e., to address one or more issues associated with a compromised part of the system), there are a variety of architectural tactics that may be employed to reduce exposure to risk when known vulnerabilities are identified.

4.4.3.1 Prevent Ripple Effects

Preventing ripple effects refers to tactics that seek to limit the amount of software that will have to be modified or replaced when a compromised component is identified. Here, we address two tactics in particular: information hiding and restricting communication paths.

Simply put, information hiding it means that the internal state of a software component is kept private, exposing only those data and interfaces required to enable it to accomplish its function within the system. Coupling occurs when the operation of one software component is directly dependent upon the internal data structures or implementation details of another component, rather than interacting with it through its public interface. Replacing or modifying a compromised component that is loosely coupled to other components is far less difficult and risky than replacing one which is tightly coupled.

Restricting communication paths means that each software component interacts with the minimum number of other components required to accomplish its function. More interactions with other software components mean more execution paths that are likely to break when a compromised component is replaced or modified. Thus, while information hiding is a tactic that limits the depth of coupling between components, restricting communication paths is a tactic that limits the breadth of that coupling.

4.4.3.2 Runtime Binding

Runtime binding is a family of tactics that seeks to reduce the effort required to deploy new or modified software in order to address a compromised component. Every system requires some level of configuration to run in its intended environment. These configuration choices can be made anywhere from the lowest level to the highest level. Enabling runtime binding entails architecting

the system such that these configuration choices are made at the highest possible level – i.e., as late as possible. While any form of runtime binding may incur a performance impact during the binding process, this cost can be recouped many times over when it becomes necessary to modify or replace a system component. This affords the ability to re-deploy parts rather than the entire system.

Runtime discovery involves dynamic tracking of system components while the system is operational. More specifically, resources can be added or removed without disrupting the system's overall operation. A failed component can be restarted, or a compromised component can be removed and replaced without bringing down the rest of the system. This kind of flexibility is extremely powerful but requires careful consideration of all the possible corner cases. Each component must be able to properly and cleanly handle the sudden disappearance of a component with which it is interacting. Otherwise, a single failed or removed component can cause all related components to fail or lock up, potentially kicking off a cascade effect that renders the entire system inoperable.

5 Secure Coding Requirements

Secure coding requirements should incentivize developers to create reliable, secure systems via several means. Coding processes that include security encourage developers to write secure code and help them find and fix security issues quickly. A thorough set of coding guidelines helps developers avoid common mistakes. Commercial, free and open source, and re-use code enables fast development, but system owners must consider security implications of using code written by another entity.

The choice of platforms and devices is often a system engineering decision, but security should be part of this decision. When choosing what platforms or devices to use, it is important to analyze what security issues may be involved with using these technologies. For example, languages such as Java that handle memory allocation reduce the risk of a large class of problems in languages such as C++, but there may be performance considerations. Additionally, as described in *The Challenge of Using C in Safety-Critical Applications* (Newton & Aschbacher, 2018), C also has very permissive semantics which can make it dangerous. Language selection and secure coding guidelines are critical component of secure software development. Also, all hardware devices and vendors should be scrutinized for security vulnerabilities using resources such as the National Vulnerability Database. (National Vulnerability Database, n.d.) The development team must create coding guidelines tailored to the hardware platform and language of choice.

Security features, especially cryptography, should be included from a trustworthy source rather than implemented from scratch wherever possible. Attempting to implement these features can be problematic, especially since there are many subtleties that, if left ignored, can lead to vulnerabilities; hence, if a proper solution has already been made by a group compliant with required standards, it is advantageous to use those instead of creating an implementation from scratch. For instance, the FIPS-compliant OpenSSL library has several cryptography functions that have been implemented to avoid issues such as timing and padding vulnerabilities. Authentication and access control mechanisms built-in to operating systems have been extensively tested and are likely to be more secure than custom implementations.

Development teams should include cyber security members starting from early concept development. Security considerations often drive the system design. For example, network defense in depth includes protections such as firewalls and intrusion detection that should be included in the network design at an early stage. As another example, strong authentication and access control will likely have major consequences for how software components interact. Adding these features late in development will be costly versus including the features in the design at an early stage.

Here is a non-exhaustive list of secure coding requirements that programs should implement:

- System engineers should choose platform, choice of language, and devices with security as a consideration. Vendors and products should be evaluated for security issues.
- Software Supply Chain Risk Management
- Developers should follow a secure coding guideline that describes best practices for writing secure code and explains security issues and how to avoid them.
- Software Bill of Materials generation and cross referencing to known adversary actions and known vulnerabilities (i.e., CVEs)
- Secure Code Guidelines
- Mission systems (e.g., Linux, Windows) should follow an accepted set of security controls and hardening. For example, CIS benchmarks, NIST 800-53, or Defense Information System Agency's (DISA) Security Technical Implementation Guides (STIGs). For software development specifically, DISA publishes an Application Security and Development STIG. MITRE also publishes Common Weaknesses Enumerations (CWEs) which can be used as a method for ensuring code is free of vulnerabilities
- Security engineers should scan commercial, open source, or re-use software with a security tool or security issues should be mitigated via mechanisms such as sandboxing.
- Developers should use trustworthy sources for security feature implementations, particularly cryptography, rather than relying on custom implementations.
- Developers should develop a prioritized list of CWEs / weakness types in which the mission wants to ensure are not within the source code
- Developers should regularly scan software with a tool or tools that provide coverage of desired security weaknesses.
- Developers should use dynamic testing which helps find issues such as memory and thread safety and timing issues. Dynamic testing methods could include day-in-the-life testing, fault management and response as well as fuzzing.
- System owners should use penetration testing to find security issues in the integrated and configured system.
- System operators should regularly scan the system for vulnerable versions of software and perform regular penetration tests during maintenance and sustainment.

For mission owners required to implement the NIST Risk Management Framework (RMF), the following NIST controls can aide in ensuring secure software development controls are in place. Many of these controls have been translated into requirement speak (i.e., shalls) in Appendix C. The intent of the appendix is to provide mission owners with sample requirement statements to put onto acquisition contracts or within the system requirement specifications.

FIGURE 8: NIST CONTROLS APPLICABLE TO SECURE SW DEVELOPMENT

NIST 800-53 Rev4

Access Control	Auditing and Accountability	Security Assessment and Authorization	Configuation Management	Contingency Planning	Identification and Authentication	Planning	Risk Assessment	Systems and Services Acquisition	System and Communications	System And Information Integrity
AC-3(2)	AU-5(2)	CA-8	CM-3(2)	CP-2(8)	IA-5(7)	PL-8	RA-3	SA-10	SC-13	SI-10
AC-4	AU-6(4)		CM-3(2)		IA-7	PL-8(1)	RA-5	SA-10(1)	SC-23	SI-10(3)
AC-4(14)			CM-4(1)			PL-8(2)	RA-5(1)	SA-11	SC-28	SI-10(5)
AC-4(2)			CM-5(3)				RA-5(2)	SA-11(1)	SC-28(1)	SI-11
AC-6			CM-7(5)					SA-11(2)	SC-3	SI-17
								SA-11(2)	SC-38	SI-2
								SA-11(4)	SC-39	SI-2(6)
								SA-11(5)	SC-4	SI-4(10)
								SA-11(6)	SC-6	SI-4(16)
								SA-11(7)	SC-7(21)	SI-7
								SA-11(8)	SC-8	SI-7(1)
								SA-12	SC-8(1)	SI-7(14)
								SA-12(1)	SC-8(2)	SI-7(2)
								SA-12(11)	SC-8(3)	SI-7(5)
								SA-12(2)		SI-7(6)
								SA-12(5)		SI-7(9)
								SA-12(8)		
								SA-12(9)		
								SA-14		
								SA-15		
								SA-15(3)		
								SA-15(4)		
								SA-15(5)		
								SA-15(7)		
								SA-15(7)		
								SA-15(8)		
								SA-19		
								SA-2		
								SA-3		
								SA-4(3)		
								SA-4(5)		
								SA-4(9)		
								SA-8		

Rev5 Changes

CM-5(3)	Moved to CM-14
RA-5(1)	Incorporated into RA-5
SA-12(1)	Moved to SR-5
SA-12(11)	Moved to SR-6(1)
SA-12(2)	Moved to SR-6
SA-12(5)	Moved to SR-3(2)
SA-12(8)	Incorporated into RA-3(2)
SA-12(9)	Moved to SR-7
SA-14	Incorporated into RA-9
SA-15(4)	Incorporated into SA-11(2)
SA-19	Moved to SR-11
SI-7(14)	Moved to CM-7(8)

Many of the new Supply Chain (SR) controls extend to software and could be included in SwA depending on the viewpoint. SR-6, SR-9 and SR-10, SR-11 are key from an assurance perspective

5.1 DEVSECOPS PROCESSES

Modern software development has embraced DevOps, or development operations, which uses automation to encourage frequent software builds, testing, and deployment. The expansion of cloud computing has accelerated the development process beyond the standard two-week Agile sprint, necessitating new processes to ensure the quality and stability of the code. Ensuring the development of secure applications has pushed DevOps to adopt new security-oriented strategies, leading to the evolutionary step of DevSecOps. The goal for successful DevSecOps is to enable the rapid development of secure code by applying the principle of "guardrails, not gates" to the development cycle. This is achieved by constantly improving the automated testing and monitoring of the system, rather than relying on a series of manual reviews.

DevSecOps is well-suited to a variety of use cases, however application of its principles can vary drastically depending on the software being developed and the environment in which it is deployed. For organizations that straddle multiple environments, SANS provides the following advice- "Organizations can use Agile or DevSecOps methods for traditional applications, however the waterfall development cannot be used for cloud-native/microservices applications- Agile or DevSecOps is essential" (Bird & Allen, 2018). DevSecOps is gaining popularity for ground system capabilities, while the application of it on the spacecraft currently lags but is not out of the question. Digital twin technology evolution may enable DevSecOps for spacecraft, but it is currently in the early stages of research. There is no prescriptive recipe for the proper implementation of DevSecOps concepts and should be adapted to each mission and system. The following section provides insight into the philosophies that guide successful adaptations of the method, as well as an introduction to the tooling required to get started.

5.1.1 FUNDAMENTAL GOALS

At the core of DevSecOps is the concept of shared responsibility across stakeholders. By combining the responsibilities of development, security, and operations staff into one cohesive group, all stakeholders take a vested interest in consistently improving their system. Developers

are incentivized to make their code easier to maintain and operate, operations staff gains a deeper understanding of the system and the technology drivers behind features, and security staff shape processes to emphasize efficiency, transparency, and a consistent application of security principles. Some organizations have transitioned to the idea of "NoOps", simply assigning each product a developer team that is responsible for all aspects of its continued operation- security and all. This may be an extreme, however it illustrates the desire to establish a comprehensive and cohesive perspective along the entire Software Design Life Cycle.

By removing many of the traditional roadblocks to deployment, DevSecOps teams are able to push new features faster by making smaller, incremental improvements. While change naturally introduces risk to any environment, making smaller changes reduces the "blast radius" of resultant negative impacts while giving the team more opportunities to improve the deployment process. Some environments are more forgiving to failures than others, but the practice of rigorously improving both the system itself, as well as the processes around the development is a vital aspect of DevSecOps.

To understand the effect of more rapid improvements, each environment must be intensely monitored. Making numerous changes to any system without tracking potentially relevant metrics, even when those changes have been tested, undercuts the ability for the team to respond to unforeseen consequences. When judging the overall effectiveness of a DevSecOps team, it is the ability to return a system to functionality after a failure- or *mean time to recovery*- that is the key capability. (Murphy, Petoff, Jones, & Beyer, 2016) A team that can identify the root cause of an issue, correct the code, automate the testing of the change, and quickly deploy the fix to production is demonstrating the key tenets of the DevSecOps method.

5.1.2 Code Repositories & CI/CD Pipelines

Missions may choose to adopt DevSecOps practices slowly, as they learn what aspects of the method should be prioritized and which may not be relevant to their use case. The use of code repositories and continuous integration/continuous deployment (CI/CD) pipelines are necessary to create the collaborative and automated environments that define the method.

System code must be stored in a central code repository with version control- through implementations of tools like Git, Subversion, or Mercurial. This allows for all members of the team to work on current versions of the code at once and enables code branches for the organized development of new features. Code repositories are essential in order to cultivate automation, with the goal to store as much of the infrastructure as code as possible. Application code, unit tests, database schemas, build/deployment scripts, documentation, and anything else necessary to build and operate the system should be checked into this central repository as possible. It should be noted that while the repository should encompass all aspects of development and testing, any form of data used for authentication- such as passwords and keys- should never be uploaded. The benefits to a comprehensive repository are numerous and are often essential to postmortems for root cause analysis and other aspects of forensic analysis. Development activities within the repository can be considered part of the Pre-Commit stage.

When team members check-in code to the repository, it typically triggers an evaluation of the code via a CI/CD pipeline. CI/CD pipelines, such as Jenkins, AWS CodeCommit and CodeDeploy, and Bamboo, are in many ways the central hub of a DevSecOps deployment, as they provide the

framework to allow development, testing, and deployment to flow end-to-end in an automated fashion. Tests within a pipeline are divided into two logical stages- Commit and Acceptance.

5.1.2.1 THE COMMIT STAGE

Commit begins with a change made to the state of the project- most often an update to the code in the source code repository. Code is compiled (if necessary) and tested to ensure that the system works at a technical level. To ensure that the code performs as expected, unit tests are performed. Once the code passes unit testing, automated code analysis tools are deployed to ensure that the committed code is of acceptable quality and security. The code analysis tools should have the capability to automatically scan for desired secure coding guidelines in addition to specific weakness classes (i.e., CWEs). Due to the diversified responsibilities of code analysis tools (quality, security, adherence to coding guidelines), it is often necessary to utilize a complementary set of tools to adequately meet the code analysis goals. Often times developers will choose tools without considering the strengths, weaknesses or objective of the tool as their requirement was to run "static analysis" which can left critical defects within the source code.

CI/CD pipelines are responsible for integrating and orchestrating the tools necessary for testing while providing a central interface to analyze test results. Analysis tool aggregators provide a method to combine results from multiple tools into a single interface. Whether using aggregation or multiple interfaces, these tools can be used to set thresholds to automate pass/fail decisions on a per-tool basis to minimize the amount of downtime spent waiting for testing to conclude. It is important to note that certain tools may assign a criticality rating to any negative finding. Criticality rating is an important feature for tailoring a CI/CD pipeline to meet the security demands of a specific mission or environment. If the developer performed the necessary rigor in the early stages to determine which weakness classes (i.e., CWEs) are of importance to the mission, then the criticality ratings will have more applicability or meaning to the mission.

5.1.2.2 THE ACCEPTANCE STAGE

After the system passes the commit stage, acceptance testing is performed to prove that all functional and nonfunctional requirements are met. This stage takes a broader approach to isolate both the operational ability of the system and the impact that deploying the current build might have on the operational environment. Therefore, the environment used for acceptance testing should closely mirror the production environment. Mirroring allows for more meaningful results and enables a mission to automate portions of its configuration management across development, testing, and operational environments. For ground systems, leveraging the advancement in virtualization/containerization, missions have been able to reap the benefits of DevSecOps. Spacecraft have yet to reap such benefits, but the advancement of digital twins via instruction set simulators DevSecOps should be a possibility soon provided the mission owners are willing to accept the risk of this new application of DevOps.

Elements of Commit and Acceptance testing can overlap, depending on the requirements of the system or environment. When first building the pipeline, however, it is advisable to methodically build additional testing in parallel based on the feedback and evolution of the system, rather than deploying an unforgiving series of tests and scans that delay and frustrate the DevSecOps team. Therefore, testing matures at a similar rate the target system, and in a manner tailored precisely for its configuration and environment. By the time the capability is ready for operational stage, the

pipeline should be a mix of rigorous automated testing, configuration and compliance checks, and manual review before changes are propagated to users.

5.2 Secure Coding Guidelines

There are many useful sources from both government and industry from which to derive secure coding guidelines. There are coding standards, including security, published by various organizations across the world. Coding standards generally are language specific (i.e., CERT C, MISRA, ISO/IEC TS 17961) whereas best practices are often language agnostic. Regardless on the nomenclature used, it is important to have standards/guidelines and having an automated method to test adherence. The following are only examples of secure coding guidelines/standards: The Software Engineering Institute's Computer Emergency Response Team (CERT) publishes secure coding guidelines for most major languages (CERT, 2016) (CERT, 2017) (Long, Mohindra, Seacord, Sutherland, & Svoboda, The CERT Oracle Secure Coding Standard for Java, 2011). Oracle publishes secure coding guidelines for the Java language (Oracle, 2017). The Open Web Application Security Project (OWASP) provides guidance for web application security (Open Web Application Security Project (OWASP), 2017). Appendix B provides some additional high-level design/coding guidelines.

Additionally, the MITRE Corporation publish common software and system security weaknesses in a database of Common Weakness Enumerations (CWEs) (MITRE Corporation, 2018). These are not necessarily coding guidelines, but they are useful for finding and learning about common weaknesses that show up in software systems.

The following sections describe several categories of software security issues to avoid which can form the basis of secure coding. These have been compiled from several sources including subject matter experts' experience evaluating many space and missile systems, a set of top CWEs affecting space systems identified by NASA's Independent Verification & Validation Program, and contractor secure coding guidelines (with proprietary information removed). They are organized into major categories of weaknesses. The prioritization and categorization performed by NASA's Independent Verification & Validation Program is available in Appendix A where they identified high priority CWEs (i.e., 346 out of over 900) provided a breakdown of domain applicability for either the ground system, spacecraft, or both. The below explanations include both summaries of the issue from the CWE and subject matter experts' interpretation with respect to space systems. Wherever possible, references to CWE categories or specific weaknesses are provided to enable more in-depth research.

The CWE category or specific weaknesses referenced throughout the document are listed in the following table.

Table 3 Top CWEs for Space Systems

CWE NUMBER	CWE Category or Weakness
CWE-19	Data Processing Errors
CWE-20	Improper Input Validation
CWE-21	Pathname Traversal & Equivalence Errors
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

CWE NUMBER	CWE Category or Weakness
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-113	Improper Neutralization of CRLF Sequences in HTTP Headers
CWE-119	Improper Restriction of Operations within the bounds of a Memory Buffer
CWE-121	Stack-based Buffer Overflow
CWE-122	Heap-based Buffer Overflow
CWE-125	Out-of-bounds Read
CWE-134	Use of Eternally-Controlled Format String
CWE-170	Improper Null Termination
CWE-189	Numeric Errors
CWE-195	Signed to Unsigned Conversion Error
CWE-242	Use of Inherently Dangerous Function
CWE-243	Creation of chroot Jail Without Changing Working Directory
CWE-252	Unchecked Return Value
CWE-264	Permissions, Privileges, and Access Controls
CWE-345	Insufficient Verification of Data Authenticity
CWE-361	Time and State
CWE-367	Time-of-check to Time-of-use (TOCTOU)
CWE-377	Insecure Temporary File
CWE-399	Resource Management Errors
CWE-416	Use After Free
CWE-421	Race Condition During Access to Alternate Channel
CWE-442	Web Problems
CWE-444	Inconsistent Interpretation of HTTP Requests
CWE-457	Use of Uninitialized Variable
CWE-465	Pointer Issues
CWE-476	NULL Pointer Dereference
CWE-477	Use of Obsolete Function
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
CWE-644	Improper Neutralization of HTTP Headers for Scripting Syntax
CWE-667	Improper Locking
CWE-676	Use of Potentially Dangerous Function
CWE-681	Incorrect Conversion between Numeric Types
CWE-682	Incorrect Calculation
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-770	Allocation of Resources Without Limits of Throttling

CWE NUMBER	CWE Category or Weakness
CWE-787	Out-of-Bounds Write
CWE-798	Use of Hard-coded Credentials
CWE-833	Deadlock
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
CWE-840	Business Logic Errors
CWE-1006	Bad Coding Practices
CWE-1021	Improper Restriction of Rendered UI Layers or Frames

5.2.1 Improper Input Validation (CWE-20)

In accordance with secure coding guidelines, input validation is a way to ensure that code cannot be manipulated in a way that was not intended, like breaking out of the norms of what the programmer expects the user to stay within. This attack can range from exposing a web application's configuration files, revealing potential attack vectors, or exfiltrating password hashes and credentials from "secure" code (Scholte, Balzarotti, & Kirda, 2012).

There are multitudes of possible vectors that an attacker can use in exploiting improper input validation, leading to denial-of-service attacks and reading or writing secure data from memory or storage, or even executing arbitrary commands on the host operating system. In order to mitigate these vectors, the code must provide robust input sanitation in any area of the application that receives input from the user.

A common vector used to attack improper input validation is the use of injection, which creates a formatted statement that cause the code to interpret the user input as an unintended operating system command or operation, outside of what the application is intended to perform. For instance, in SQL statements, without proper input validation, it is possible to manipulate any statically defined SQL statements in code to be interpreted as a comment, and instead execute custom statements with malicious intent (CWE-89). In OS command injection, it is possible to insert a malicious operating system command, with the intent to exfiltrate secure data from the host system.

In this PHP example code:

```
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

The user input for the variable $userName is not sanitized or verified before executing it on the

host system as a parameter to the PHP syscall function. As a result, it is possible to append a semicolon to the user's input to expose the system's password hash file:

```
; cat /etc/shadow
```

This will allow the system to first process the ls command, then print out the password hash file as a consecutive command. While there are other guidelines for preventing command injection at a host level, there are also mitigations that can be taken at the application code level.

SQL injection functions similarly to OS command injection, in that an unintended database management system command can be run, and the attack vector is also usually in the form of user input. If an attacker can determine that an application is using SQL calls, they may be able to manipulate hard-coded SQL statements to execute unwanted actions, possibly exfiltrating secure data or corrupting critical data, bypassing authentication measures, or using information about the database to gain privileged access to the host system (Halfond, Viegas, & Orso, 2006).

For most application features, it is not necessary to directly call an operating system host command, and rather utilize a library that is able to produce similar results, in order to avoid exposing access to the underlying operating system (MITRE Corporation, 2018) (CWE-20).

At the operating system level, it is possible to limit the application to a sandbox, where the access to critical files or commands is limited. This can be achieved through the use of solutions such as chroot, AppArmor, SELinux, or alternatively, containerizing the code with Docker in order to further limit access to the host system.

If fine-grained input is not strictly necessary from the user, it is possible to mitigate command and SQL injection by only providing a secure and vetted set of inputs for the user. If direct user input is necessary, the user's input should be parsed and searched for characters that could indicate malicious intent, such as semicolons or redirection characters, as well as some comment characters in the case of SQL. In the more specific case of SQL, it may also be necessary to include a library that specifically handles SQL statement execution, which is able to only parse intended statements against the database server.

5.2.2 Permissions, Privileges, and Access Control (CWE-264)

The three A's in access control are authentication, authorization and accountability. Authentication is proving identity, authorization is proving the right to access a resource, and accountability is logging user activity. Any failure within these three portions of access control will allow an adversary to read, modify and remove sensitive data, execute programs, and evade detection. Failure can occur due to incorrect specifications (privileges, permissions, ownership) set by the administrator or by a program, or due to bugs within the access control program that prevent it from enforcing security policies.

Common pitfalls for incorrect specifications include: insecure default permissions set by a program, multiple objects having insecure permissions inherited from one object within the program and incorrect preservation of permissions when copying, restoring, sharing or executing objects. Additionally, the program must have a boundary, often called "sandbox", wherein

sensitive data is held; this boundary must be tested to have full privilege separation functionality before production use.

In a Unix environment, chroot() is a system call used to limit access of a program. If used incorrectly, access control will be void. Consider the following code snippet:

```
chroot("/var/ftproot");
...
fgets(filename, sizeof(filename), network);
localfile = fopen(filename, "r");
while ((len = fread(buf, 1, sizeof(buf), localfile)) != EOF) {
       fwrite(buf, 1, sizeof(buf), network);
}
fclose(localfile);
```

Source: (MITRE Corporation, 2018) (CWE-243)

This program is used to transfer files between computers. Because the working directory isn't changed, the program isn't limited to the directory set by chroot; an attacker would then have free reign to access any sensitive file on the computer, including the password file stored on the machine.

Dangerous functions may be improperly restricted, allowing an attacker to access them. This generally occurs when a function was never intended to be accessed by an outsider, or when a function was meant to have limited access, but the developer either did not implement restrictions or did so incorrectly. This issue can be mitigated with proper design prior to implementation wherein functions that must be exposed and the persons to whom those functions must be exposed to are identified and the need for them to be accessed validated; if methods must be exposed, all arguments must have input validation and authorization must be limited.

An often-neglected form of incorrectly implemented access control is a lack of encryption. Sensitive data left in cleartext can be read by an attacker. Sensitive data could be login credentials stored in a cookie, in an environment variable, the registry, in a program's properties file or in the program executable itself. Even if the data is not human readable, certain techniques can allow an attacker to view this data, hence only FIPS-complaint encryption should be used.

Access control must be coupled with the practice of least privilege, the principle requiring that users, processes, and programs may only access information and resources necessary for their legitimate purposes with the least privileges required. When an administrator account is required, however, it is important to limit administrator privileges to a small subset of users, or even restrict the administrator accounts on a function or time-limit basis. To keep these administrator accounts secure, a two-factor authentication system—such as a token or a smartcard—must be implemented to add a failsafe in case a password is cracked or otherwise acquired by an adversary.

Passwords should never be encrypted or stored in plain text. It's recommended to abide by the NIST Special Publication 800-63B. Section 4.1 of the guidelines state that passwords must be

salted with random 32 bits and hashed using PBKDF2 with at least 10,000 iterations. Password complexity should include ASCII special characters, numbers, lowercase, and uppercase letters. Cookies should not include the user's password and should instead include a new password generated by the session host with at least 64 bits of entropy. For more information, see section 7.1 of the guidelines. Credentials should never be hard-coded, or otherwise built-in, to any software or website. Password reuse should be forbidden, though many employees ignore this rule. This neglect in combination with hard-coded credentials may allow access to systems beyond the one with hard-coded credentials (MITRE Corporation, 2018) (National Institute of Standards and Technology, 2017) (CWE-798).

5.2.3 Data Processing Errors (CWE-19)

Errors in data processing can lead to vulnerable applications, as these open common attack vectors that malicious actors target first when assessing an application.

In general, this class of errors deals with input provided by the application user, and thus, command or other types of injection can be performed on the host system through code that lacks input validation (see 0).

In the case of format strings, an entire set of exploits are possible for a program, especially if an initial instance of a format string vulnerability is copied across the entire codebase. This occurs primarily in C, since it allows the code to accept as many arguments as possible, thus, if the code uses an unsafe function that allows the parsing of format strings with user input, this is a possible attack vector. A malicious attacker can exploit unsafe string parsing to execute shellcode.

Though intentionally crafted, consider the following code snippet:

```
#include <stdio.h>

void printWrapper(char *string) {

       printf(string);
}

int main(int argc, char **argv) {

       char buf[5012];
       memcpy(buf, argv[1], 5012);
       printWrapper(buf);
       return (0);
}
```

Source: (MITRE Corporation, 2018) (CWE-134)

The use of a printf() call in this example code allows for a wide-open attack vector, in combination with the large buffer size, which allows for virtually any sort of shellcode execution.

C is the most vulnerable language to this error, as it is not type-safe, and while some other languages may be vulnerable to a format string attack, it is likely not as effective as running the attack on an application written in C. Therefore, if possible, the most overarching mitigation for this category of issue is to rearchitect the code to use a type-safe language instead. However, if this is not possible, format string vulnerabilities can usually be detected at compilation time by the compiler or through static analysis.

In other cases, mishandling or abusing sentinel or null terminator characters can cause unexpected logic errors at runtime. This error generally involves string processing in C, which can be caused by the insertion of a sentinel character, usually in the form of a null terminator, as a part of user input. Although this does not open any serious attack vectors as format strings, sentinel characters may cause a loss of data integrity by truncating data in unexpected areas.

However, as a related data error, the misuse or omission of a null terminator character can have serious side effects, allowing for potential access of memory beyond the allocated area or crashing a system by overflowing a buffer. In other cases, exploitation of a null terminator attack vector can allow an attacker to take control of a host system through code execution.

This form of attack is most seen in C, which strictly requires the correct usage of a null terminator for proper code functionality. Even with the "safe" versions of functions that are designed to have safeguards that traditionally "unsafe" functions lack, they can still be vulnerable to a null terminator-based attack like the following code example:

```
#include <stdio.h>
#include <string.h>

int main() {

       char longString[] = "String signifying nothing";
       char shortString[16];

       strncpy(shortString, longString, 16);
       printf("The last character in shortString is: %c (%1$x)\n",
              shortString[15]);
       return (0);
}
```

Source: (MITRE Corporation, 2018) (CWE-170)

In this case, the data handled must be considered carefully, because there is no null terminator placed at the end of character array/string shortString, even with the use of strncpy(). It is also possible to use this exploit in non-C based languages, such as PHP, where null termination can effectively be used to exfiltrate data from non-web application directories, even if other traditional input validation issues are mitigated:

```
$file = $_GET['file'];
require_once("/var/www/images/$file.dat");
```

Even though this PHP snippet strictly requires a file with a predefined extension, it is still possible to use a null terminator attack to terminate the string at the $file variable indicator, and instead access any file from the host system's filesystem.

In some cases, data errors can branch into more subtle code issues, such as unintended exposure of secure data. This will occur if debug statements are used at any point in the code, such as in catching exceptions or in some cases where an attacker may be able to crash a web server, application, or database management system to reveal underlying, built-in debug routines (Carnegie Mellon University, n.d.). In many cases, system information may be disclosed even with normally functioning code.

A wide range of potential host system information can possibly be revealed, such as web application filesystem paths, stack traces from exceptions, detailed exception information, or general system information that may include kernel versions. Even based on this limited information revealed, an attacker may be able to compile enough data on a host system to exploit it further. The best way to mitigate this class of data processing error is to configure the web application or server to redirect all logs and error output to a secure log file, inaccessible by normal means from the web application.

A further class of data processing errors generally consists of logic-based errors in coding, but may lead to a loss of integrity, as casting in code may cause a loss of precision. Though some language may be able to handle type casting gracefully, in many languages, it may cause a loss of data precision if converting data between types without verification.

In the PHP code sample below, when adding the two variables together, casting the floating-point value to an integer will round it down, which may be an unexpected result as the value is 4 instead of 5. To best mitigate this data processing error, avoid casting where possible and keep the use of primitive data types consistent. Additionally, the implementation of floating-point values may not be portable across languages and architectures (Hauser, 1996).

```
$floatVal = 1.8345;
$intVal = 3;
$result = (int)$floatVal + $intVal;
```

Source: (MITRE Corporation, 2018) (CWE-681)

5.2.4 Numeric Errors (CWE-189)

Though innocuous at first glance, a numeric error in code can have somewhat far-reaching consequences and lead to a possible attack vector for a malicious actor. In general, the class of numeric errors involve simple oversight in performing memory accesses or calculations of variables used in code but can lead to a denial of service due to unexpected results, possible

resource exhaustion or opening of a memory-based exploit in memory allocation numeric errors, or memory corruption and exposure.

In the case of incorrect calculation, the aforementioned consequences open up, with the additional possibility of exploiting them as an attack vector into a host system. Incorrect calculations can lead to overflow depending on the datatype used, causing unexpected output or runtime errors. In other, more severe cases, an incorrect calculation can lead to serious memory-based errors or vulnerabilities, such as in the following example:

```
int *p = x;
char * second_char = (char *)(p + 1);
```

Source: (MITRE Corporation, 2018) (CWE-682)

In this example, the memory offset used to reference the integer will cause an incorrect offset. Although the effects of this incorrect calculation are somewhat dependent on the actions taken to the variable second_char and the system architecture, it still has the ability to cause reads or writes on unintended areas of memory.

Incorrect calculations can be primarily mitigated by carefully considering the programming language's compiler's interpretation of the code, as well as the host system's architecture, which can assist in preventing flawed code such as in the above example.

For cases of overflow or unexpected results, validation should be utilized to make sure that intended values do not overflow, and stay within a certain bound, or consider data types or libraries that are intended to prevent overflow or out-of-bounds logic errors. Additionally, in the case of incorrect calculation with dynamic input, it may be possible to use a fuzzer or other dynamic testing tools (see section 6.2) to find input that causes the application to behave unexpectedly through incorrect calculation errors.

Another subset of numeric errors is in integer coercion errors, otherwise known as improper type conversion of primitive data types. While not as devastating as incorrect calculations, integer coercion errors can have an impact on integrity and availability, and rarely on confidentiality if the improper type conversion leads to a buffer overflow vector.

Consider the following code snippet:

```
DataPacket *packet;
int numHeaders;
PacketHeader *headers;

sock=AcceptSocketConnection();
ReadPacket(packet, sock);
numHeaders =packet->headers;

if (numHeaders > 100) {
       ExitError("too many headers!");
}
headers = malloc(numHeaders * sizeof(PacketHeader);
ParsePacketHeaders(packet, headers);
```

Source: (MITRE Corporation, 2018) (CWE-195)

In this example, the application receives a user-defined number of packet headers, which is then verified and allocated in memory; however, the input is initially stored as a signed integer. A malicious user can input a negative number that bypasses the initial range check, but when the system attempts the memory allocation, a negative integer will overflow the datatype used by malloc to determine the amount of memory to allocate. This be manipulated to either attempt to allocate too much memory, causing a denial of service, or too little memory, possibly opening a vector for a buffer overflow attack.

The solution for integer coercion errors can be mitigated at the coding language level, by alternatively using a language that will catch and throw exceptions upon encountering a coercion error. If this is not a feasible task, the code should be refactored in order to prevent coercion from being necessary, or if strictly required, carefully consider the primitive data types in use to prevent overflow from causing the previously mentioned issues.
Another subset of numeric errors is the incorrect calculation of buffer size, which can be an extension of an incorrect calculation or integer coercion errors. The root cause is the same, where an incorrect calculation causes the buffer size to be incorrect, which may either cause a buffer of unexpectedly small size, opening a buffer overflow vector, or in the case of an overly large buffer allocation, the application may crash on attempting to allocate the memory. This error can still occur even if the application is coded according to a known set of expected input ranges and would behave normally in most circumstances.

In the sample code below, assumptions are made about the character encoding scheme, and thus an implicit buffer size calculation is made. With the addition of the ampersand character to the destination buffer variable, the encoding scheme will now require more memory allocated (five times the size of a C character rather than 4). However, since the code statically allocates memory, it is possible to overflow the buffer via inputting too many ampersands.

```
char * copy_input(char *user_supplied_string){
	int i, dst_index;
	char *dst_buf = (char*)malloc(4*sizeof(char) * MAX_SIZE);
	if ( MAX_SIZE <= strlen(user_supplied_string) ){
		die("user string too long, die evil hacker!");
	}
	dst_index = 0;
	for ( i = 0; i < strlen(user_supplied_string); i++ ){
		if( '&' == user_supplied_string[i] ){
			dst_buf[dst_index++] = '&';
			dst_buf[dst_index++] = 'a';
			dst_buf[dst_index++] = 'm';
			dst_buf[dst_index++] = 'p';
			dst_buf[dst_index++] = ';';
		}
		else if ('<' == user_supplied_string[i] ){

			/* encode to &lt; */
		}
		else dst_buf[dst_index++] = user_supplied_string[i];
		}
		return dst_buf;
}
```
Source: (MITRE Corporation, 2018) (CWE-119)

In general, incorrect buffer size calculation is a wide-ranging issue, and the best way to course of action to prevent this from affecting an application is through careful consideration of the datatypes involved, as well as not making assumptions on the type of data a potential user will input.

An example of a real-world consequence resulting from buffer size miscalculation is the Heartbleed vulnerability, which was the result of not realizing the true size of heartbeat packets, leading to a massive buffer overflow attack vector. This attack vector allowed attackers to read past allowed memory on the server through crafting a heartbeat response request larger than the intended length.

To avoid such grave errors, the overarching solution is to rearchitect the application to not use C code, as many other modern languages handle memory allocation to prevent buffer overflows. If C is strictly necessary, use non-array data structures that are more robust and allow for better association between data accesses and valid memory, such as graphs or queues (Black & Bojanova, 2016).

5.2.5 Improper Restriction of Operations Within the Bounds of a Memory Buffer (CWE-119)

Buffer overflows and misusage with memory read and writes prose critical vulnerabilities and risk to a software system. In 2014 a single buffer over-read in OpenSSL allowed hackers access to sensitive data such as private keys via crafted packets. This vulnerability became to be known as Heartbleed, which poses as just one of the many security bugs associated with improper handling of buffers (National Vulnerability Database, 2014). These types of errors have the potential to bypass authentication, retrieve passwords, crash systems, and cause covert code execution.

When creating a memory buffer, ensure that the program is writing to the intended place in memory, otherwise an out-of-bounds buffer write could occur. This entails a situation where the buffer writes data past or before the beginning of its allocated slot of memory (CWE-787). Memory overwriting can occur within heap (CWE-122) or stack memory (CWE-121) as well. A heap-based overflow typically occurs when functions such as malloc(), memory allocation function within C, are misused. This situation can cause functions and data stored within the heap to be overwritten, opening a system up to memory exploitation. (Ferguson, 2007)

In the C example below, the buffer within the malloc() is given a fixed size within the heap, but the argv[1] value within the strcpy method could exceed the allotted value, resulting in an overflow. This situation can become exploited to overate memory within the heap if the argv[1] was purposefully given a value exceeding the buffer size.

```
#define BUFSIZE 256
int main(int argc, char **argv) {
       char *buf;
       buf = (char *)malloc(sizeof(char)*BUFSIZE);
       strcpy(buf, argv[1]);
}
```

Source: (MITRE Corporation, 2018)

Stack-based overflows occur when a buffer allocated to the stack is overwritten. These situations can result in function parameter values being changed, as well as local variables within a program. The example in C below is similar to the one above, but notice it lacks a malloc(), a function that is associated with the heap. Within this code, the argv[1] value is not checked before being parametrized within strcpy(). Just as the heap example, this variable can be purposely overloaded to overwrite data within the stack.

```
#define BUFSIZE 256
int main(int argc, char **argv) {
       char buf[BUFSIZE];
       strcpy(buf, argv[1]);
}
```

Source: (MITRE Corporation, 2018)

Other than overflows, buffer out of bounds issues can occur due to semantic error (CWE-125). When memory is pointed to a position before the buffer, memory can be accessed past the buffer. This situation occurs when an index or a pointer.

When managing memory operations using buffers, always ensure logic is sound and all measures are being followed to prevent buffer out of bounds errors. Memory reads and writes should be fixed and checked to the exact value that is being read or written.

5.2.6 Logic Errors (CWE-840)

Logic errors may be a significant time sink in secure coding assurance, as they still allow the program to execute after compilation but may produce unexpected behavior. This category of errors can impact an application's availability and confidentiality.
In the case of many applications, it is not necessarily feasible to keep all application-critical data in the application's allocated memory, as it may be insufficient, especially if the data is associated with multiple client connections. As a result, the code may be designed to frequently read or write files to disk, which may involve also setting permissions to ensure correct availability of files only for authorized host system users. In this case, the code must be assessed to ensure that correct permissions are being set on an accessed file, as in most cases, the file will initially inherit the parent process's permissions, but may unpredictably change following subsequent application execution (MITRE Corporation, 2018) (Seacord, n.d.) (CWE-732). .

In the Perl code sample below, the output file created may inherit permissions from its parent process or running user and will be created with restrictive permissions. However, if this file is reused across a modular application or by other users on the host system, the permissions may change unexpectedly, and become world readable and writable due to the highlighted snippet, which is likely not the intended behavior or availability desired.

```
$fileName = "secretFile.out";

if (-e $fileName) {
       chmod 0777, $fileName;
}

my $outFH;
if (! open($outFH, ">>$fileName")) {
       ExitError("Couldn't append to $fileName: $!");
}
my $dateString = FormatCurrentTime();
my $status = IsHostAlive("cwe.mitre.org");
print $outFH "$dateString cwe status: $status!\n";
close($outFH);
```

Source: (MITRE Corporation, 2018) (CWE-732)

To prevent this form of logic error from occurring, the application should be coded to a checksum on the file to verify the lack of tampering with respect to the previous run, as well as ensure that permissions have not been altered to an unexpected value. At the host system level, the application

can be isolated from the rest of the system through containerization, or permissions can be set on specific files to ensure they are immutable.

While incorrect permissions are an instance of a logic error that can impact confidentiality, infinite loops caused by faulty logic in code can critically impact an application's availability. A loop in application code logic is caused by unexpected conditions being provided, causing the programing to continuously execute a set of instructions, consuming vital resources in CPU cycles or memory, and locking the application in a state where it may not be able to respond to further incoming requests.
See the following code snippet:

```
public boolean isReorderNeeded(String bookISBN, int rateSold) {

       boolean isReorder = false;

       int minimumCount = 10;
       int days = 0;

       // get inventory count for book
       int inventoryCount = inventory.getIventoryCount(bookISBN);

       // find number of days until inventory count reaches minimum
       while (inventoryCount > minimumCount) {

              inventoryCount = inventoryCount - rateSold;
              days++;
       }

       // if number of days within reorder timeframe

       // set reorder return boolean to true
       if (days > 0 && days < 5) {
              isReorder = true;
       }

       return isReorder;
}
```

Source: (MITRE Corporation, 2018) (CWE-835)

In this case, the code will run as expected, unless the provided value for rate sold is zero or negative, as the code will become stuck in the while loop, either unable to decrement the inventoryCount variable as expected, or in the case of a negative value, loop until inventoryCount

overflows. For both scenarios, the while loop instructions execute continuously, and lock resources until the application is terminated or the data overflows.

The best solution to mitigate logic errors causing infinite loops is to implement input validation or impose limits within the code, to prevent loop-based instructions from executing continuously. In the above example code, validating the rateSold parameter to be greater than 1 will prevent the code from being stuck in a loop.

Another possible situation where limits must be imposed to prevent infinite loops is if the application attempts to make a connection or wait for a response from a client application. In this case, the logic may allow for the server application code to wait indefinitely, executing a looping instruction that waits for a client response.

5.2.7 Pathname Traversal Errors (CWE-21)

Depending on application requirements, the software developer may allow the application direct access to files on the host system, and therefore may be coded to allow relative or absolute paths in order to find the required files. However, if user input is allowed at any point involving path traversal, it may be possible for a malicious actor to break out of the intended filesystem locations and access confidential data.

This potential attack vector has been the source of many acknowledged bugs in commercial software and most commonly found in web applications or servers; one notable example is in Microsoft's ASP.NET (Burnett, 2004). The errors in path traversal allow attackers to break out of the traditional web server hierarchy to access protected files outside of the host system's web directory. The most common form of this attack on a web application involves using special relative directory references with "..", which is a relative reference to the parent directory. If the application or web server is not properly coded, it may allow this path traversal.

Once an attacker confirms that an application or web server is vulnerable to pathname traversal errors, they can exfiltrate data or alternatively overwrite files with malicious ones. For instance, if the application is hosted on a Unix-like system, the attacker may be able to insert a new password entry to the end of the encrypted passwd/shadow file to obtain full shell access. Alternatively, the attacker can also overwrite known system binaries or libraries with malicious content, such as keyloggers, or nonfunctional code that will cause a denial of service.

For instance, in the following example Perl code:

```
my $dataPath = "/users/cwe/profiles";
my $username = param("user");
my $profilePath = $dataPath . "/" . $username;

open(my $fh, "<$profilePath") || ExitError("profile read error:
$profilePath");
print "<ul>\n";
while (<$fh>) {
	print "<li>$_</li>\n";
```

```
}
print "</ul>\n";
```

Source: (MITRE Corporation, 2018) (CWE-22)

In this snippet of Perl code, the application accepts a username from the user as a parameter, which is then accessed directly from the host's filesystem. However, since there is no path validation in this case, if an attacker uses relative parent directory references, it may be possible to cause the code to open a secure file, such as the passwd or shadow file. For instance, if the above code is provided the following output:

```
../../../etc/shadow
```

The input string shown will cause the application to traverse into the previous relative up to the root, then open the secure hash record file for the host system, instead of creating a user directory.

There are wide ranges of possible mitigations to take in response to pathname traversal errors. One consideration is to rearchitect the application, and not allow input from the user that could be used for such path traversals outside of the intended files allowed. For the above example code:

```
if (index($profilePath, "/") != -1 || index($profilePath, "..")){
       print "Error: Invalid Path";
       [Return to calling function or quit script]
}
```

This if statement is only for illustration but is a possible example that will help mitigate path traversal, as it would theoretically disallow any user input that may attempt to traverse into directories containing secure data.

If user input is necessary, provide validation of the user provided input values and match against a whitelist of allowed directory values, while limiting parent path traversal by restricting the special "..". Additionally, many languages provide a function for revealing the full canonical path of a string, which can also be compared against a whitelist to prevent traversal outside of the application's intended directory.

At the operating system level, it may be possible to configure the web server itself to automatically canonize any attempted filesystem traversals and prevent files outside of the web application directories. In most cases, the web server process is run as an unprivileged user that does not have access to system files such as the passwd file, but an attacker may still be able to gain access to hidden files or other confidential data that is within the web application directory's bounds.

5.2.8 Insufficient Verification of Data Authenticity (CWE-345)

Integrity checks are a way of checking data authenticity and are a vital part of implementing safe software security measures to ensure that data has not been compromised during transfers, by human error, from hardware, or by malicious users. They are required in adherence to the National Institute of Standards and Technology (NIST) guidelines regarding security and privacy controls

on federal systems. If the integrity check is poorly designed or missing all together, then a system can accept invalid data. Software assurance measures insists that critical data within a software system should have its integrity maintained for its accuracy and consistency throughout its entire life-cycle. This may be enforced with cryptographic hashes and/or encryption/decryption. Algorithms should be implemented to verify the logical and physical integrity of data using these mechanisms (National Institute of Standards and Technology, 2017).

5.2.9 Time and State (CWE-361)

Time and state can play a huge role in security, especially when it comes to multi-core, multi-threaded, and distributed systems. Critical resources being shared across multiple processes, computers, or cores can produce vulnerabilities to a software system if they are not properly warded against race conditions with proper locking.

5.2.9.1 Race Conditions

A race condition is an undesirable situation that involves multiple operations trying to access memory or conduct the same execution at the same time. Race conditions can cause systems to crash, halt, or corrupt memory. In every instance a race condition can occur, measures, such as proper locking of resources, should be implemented to prevent a software system from deviating from desired behavior.

A Time-of-check to Time-of-use (TOCTOU) race condition describes an instance where a critical resource's state is checked, but the state can still change before the resource is used. In this vulnerability, the purpose of the check is nullified, and an attacker can influence the state of a resource during its availability (CWE-367). An attacker can gain access to this unauthorized resource and the resource can be maliciously altered, such as log files and memory. There are instances where a temporary file on a system (CWE-377) could be used to steal data during an attack (MITRE Corporation, 2018). Often, engineers may want to use a temporary file when developing a program. If left unchecked, unauthorized users can access and modify the file, which could crash a program or alter its memory.

Attackers can access a privileged resource if an alternate channel to the resource is unfiltered. In some situations, an attacker may also be able to assume identity and gain privileges to other resources (CWE-421).

Nearly all race conditions are preventable, and preventive measures can be taken during each phase of the development process. During design, it is important to determine the critical resources of a system and which components of a system require access to that critical resource. A shared resource should be evaluated to ensure its necessity to be read and updated by multiple components. If it does not make sense for the resource to be shared across a system (or systems), then the resource should stay in scope of the individual component. Mutex locks and semaphore implementations can project against race conditions, as they prevent from multiple processes accessing data at the same time.

5.2.9.2 Improper Lock Handling

Improper locking of a resource, an instance where a time and state sensitive area is not controlled with a lock, makes the resource vulnerable to race conditions (CWE-667). Locks should be used

to prevent race condition situations. In order for a thread, core, or system to access a shared resource, they will request to unlock the source if it is available for use. If not, the thread, core, or system should wait until the source is free. If this method is not implemented correctly, a myriad of security and performance vulnerabilities will arise.

A deadlock can occur as well, a situation where multiple threads, cores, or systems are waiting for a resource to be available, but there is no possible way to unlock the resource (CWE-833). This causes a program to halt or crash because all processes become stalled until they can access their resource. Excessive locking of a critical resource can also result in a program crashing since performance is degraded. An attacker could exploit this and cause a Denial of Service (DoS) attack on a system.

A possible solution for improper lock handling is to employing deadlock avoidance policies (DAPs) to implement secure locking methods (Reveliotis & Fei, 2017). Following DAPs assist in preventing dead locks, they also instruct on proper lock handling, preventing a multitude of issues regarding lock misusage.

5.2.10 Bad Coding Practices (CWE-1006)

Writing software yields itself to bugs. Some of those bugs may not be easy to detect, allowing security vulnerabilities in the software that may compromise sensitive data. To protect against these issues, the development team must be aware of and actively avoid bad coding practices. These can include using dangerous functions, ignoring compiler warnings, leaving uninitialized variables, writing bad documentation, and improperly logging errors.

CWE-477 and CWE-676 describe potentially dangerous functions that, if used incorrectly, can cause a vulnerability. For instance, in C/C++ using strcpy without checking that the source can fit into the destination can cause a buffer overflow. This can be avoided by adding checks before data is copied. It is recommended to identify APIs that can be potentially dangerous and brief developers upon their proper use. Similarly, CWE-242 describes inherently dangerous functions that cannot be used safely. Generally, these functions have no way to be validated, causing unpreventable buffer overflows. Inherently dangerous functions must be identified, logged, and prohibited. Some static analysis tools, like the Clang Static Analyzer, warn the developer about dangerous functions; such tools should be implemented into the development process.

Uninitialized variables are addressed in CWE-457. In addition to causing general instability in a program, uninitialized variables allow an adversary to launch a denial-of-service attack should they identify a way to trigger the use of an uninitialized variable. Using default variable values and setting the compiler to warn about uninitialized variables are good programming practices, even in languages that do not require explicit variable declaration. In general, developers should be advised to heed compiler warnings and compilers should be set to the highest warning levels. Simple mistakes such as uninitialized variables can cause catastrophic events but can be easily noticed if the compiler is configured properly.

Oftentimes when employees leave the company, or even a department within the same company, their knowledge leaves with them. To avoid this, all function behavior, input, and output parameters, return values, exceptions, custom APIs and frameworks must be sufficiently documented. The documentation must be comprehensive enough for a person without prior knowledge to gain a full understanding of the code.

Improper logging can cause leaking of sensitive information. For instance, a file not found error may include a file name that is sensitive. Additionally, errors that are too descriptive can allow an adversary to keep prodding at a program, using logs and exceptions as clues to aid in their attack. It's advised to keep logs from being too descriptive. Instead, use error codes only known internally and reference those codes within your logs. This will defend against adversaries using logs as clues to their attacks.

5.2.11 Resource Management (CWE-399)

Resource management of host system memory at the code and application level is critical in both maintaining the application's stability, as well as in preventing secure data from being leaked, even after the termination of the application. This is a broad category that includes misuse of system resources leading to resource exhaustion, failing to properly free memory, leading to memory leakage, attempting to free memory twice, or attempting to access a memory address after it has been freed.

Resource exhaustion can occur through multiple vectors, whether caused by developer oversight or malicious intent to cause denial of service. In general, resource exhaustion occurs when the application's code keeps the use of resources unchecked, like when accepting client connections or allocating memory for user input. If limits are not imposed on system resources, resource exhaustion will impact the host system's availability.

Consider the following code snippet:

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {
	newsock=accept(sock, ...);
	printf("A connection has been accepted\n");
	pid = fork();
}
```

Source: (MITRE Corporation, 2018) (CWE-770)

In this application, there are no limits on the number of client socket connections that can be made, and on every connection a process fork occurs. Since forking the client code creates an entirely new instance of the application for the client, an attacker can possibly create an infinite number of client connections and tax the host system with multitudes of forked processes.

Memory exhaustion can also occur when usage is not checked, and input is allowed from a user:

```
int processMessage(char **message)
{
	char *body;

	int length = getMessageLength(message[0]);
```

```
        if (length > 0) {
                body = &message[1][0];
                processMessageBody(body);
                return(SUCCESS);
        }
        else {
                printf("Unable to process message; invalid message
length");
                return(FAIL);
        }
}
```

Source: (MITRE Corporation, 2018) (CWE-770)

In this sample function, the application accepts user input as a message string, where the only verification enacted is the string length being a positive, non-zero value. However, it is possible for the string to be an extreme length and could use a large amount of system memory in storing the string.

The best way to mitigate resource exhaustion of all kinds is to implement data validation or limits on the amounts of resources allowed to be consumed by the application. At the system level, the application's system host user can be confined to a limited amount of memory or CPU usage and ensures that less resources are dedicated to application-level users. At the application code level, input validation should be enacted, to ensure that any fields requiring user input are not able to cause memory exhaustion.

A double free may occur in code due to attempts to free memory addresses that have already been freed (OWASP, n.d.). Application code should also be checked for a possible double free of allocated memory, as an attempt at a double free may cause a denial of service through causing a crash or allowing a malicious user to cause a buffer overflow and have access to the host system's memory. In general, a double free may be hard to identify because it can occur at multiple points within code, and across multiple conditional statements. The best solution is to identify potential double free occurrences with static code analysis, which should be run as a part of secure coding by default.

For all resource management concerns, if possible, consider an alternative programming language. For instance, Java, which runs all code in a virtual machine and provides its own form of memory management to prevent the host system from being overloaded.

5.2.11.1 Pointer Issues (CWE-465)

Improper use of pointers causes problems within software that ultimately leave software systems vulnerable to exploits. A NULL pointer deference is an instance where a software application points to a blank space in memory that causes processes to crash or force exit (CWE-476). After a forced abruption, returning the software program back to a safe state of operation can be difficult. This leaves an opportunity for an attacker to keep a software down and possibly cause code execution depending on some OS architectures and programming languages (MITRE Corporation, 2018). An example of this is captured below in C where an IP address is being written to the

hostname buffer within the strcpy() method. If an IP address were not in the list of hostnames, a NULL pointer could be returned, crashing the program.

```
void host_lookup(char *user_supplied_addr){
       struct hostent *hp;
       in_addr_t *addr;
       char hostname[64];
       in_addr_t inet_addr(const char *cp);
       validate_addr_form(user_supplied_addr);
       addr = inet_addr(user_supplied_addr);
       hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
       strcpy(hostname, hp->h_name);
}
```

Source: (MITRE Corporation, 2018)

Another pointer related security flaw to consider is incorrect pointer scaling, a situation where a pointer contains semantical error involving implicitly called math operations. Incorrect pointer scaling can lead to a buffer over-read or under-read (refer to 3.2.5).

Use-after-free (CWE-416) is a situation when a program returns dynamically allocated memory to the heap with a free operation, but uses the pointer as though it were still valid later. The memory may have been re-allocated for another use. This may allow an attacker to read or overwrite sensitive information at the re-allocated location.

Nearly all pointer issues are preventable with extensive testing and exception handling. Every situation when a pointer is introduced should be analyzed and determined if the given situation would benefit from using a pointer as opposed to a pass-by-value variable.

5.2.12 Web Problems (CWE-442)

Though ideal for catering towards larger populations, whether on an internal or public network, web applications are susceptible to being attacked. Due to their accessible nature, there are many features on websites that must be safeguarded from malicious users.

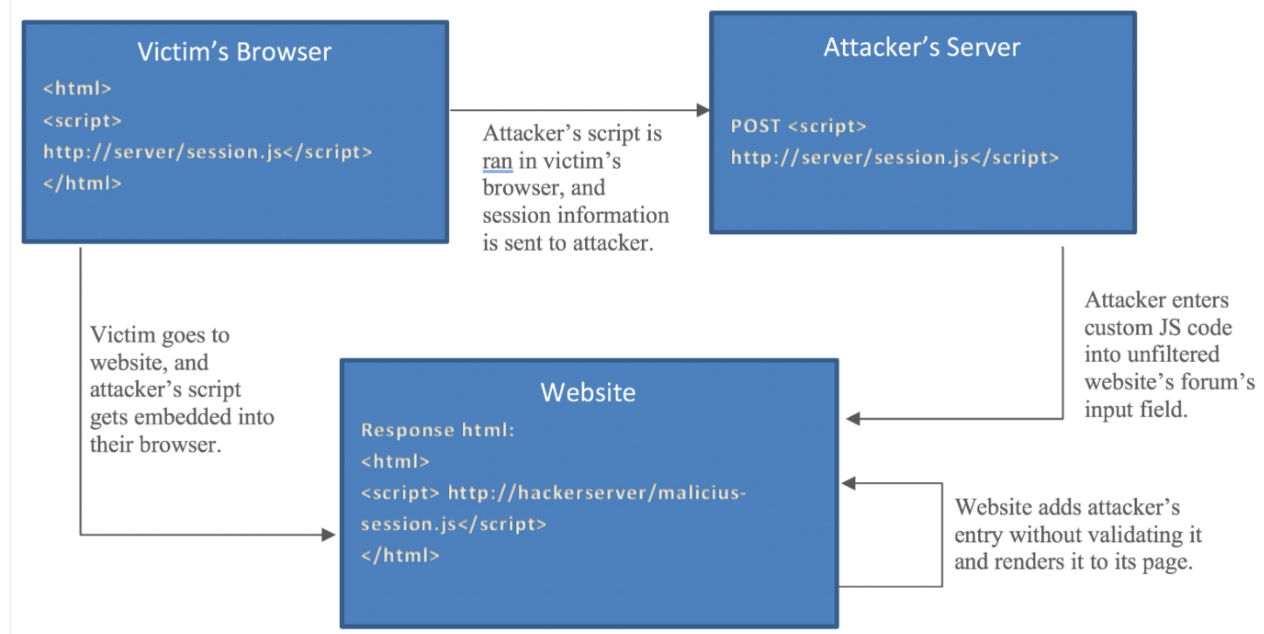
Typical website architecture relies on three main components: a frontend, a backend, and a codebase that interfaces between the two. Each component requires end-to-end security, from when a user inputs data into a text field box up until it gets stored into a database. When input is prompted by the user within frontend interface, ensure that input is validated. Sensitive information, such as hash passwords, should be stored properly within a secure database free of possible code injection (refer to 0) (CWE-1021). Account and user access should be controlled and limited, to prevent improper access to data (refer to 5.2.2).

Encrypting requests from point A to B is a secure method when approaching a web application. Encryption ensures that unauthorized, third-party listeners are unable to decode the bytes being transferred. Confidentially is one of the reasons why HTTPS is becoming a standard for websites

instead of HTTP, as all communications between the website and the web browser are encrypted(CWE-444) (CWE-644) (CWE-113).

Cross-site scripting (XSS) is a vulnerability that allows non-developer code to be injected into a web browser executable code (CWE-79). XSS enables an attacker to directly interact with a website to exploit it or put malicious code on a victim's browser (CWE-601). Below is an example of a potential XSS exploit that runs a script from an attacker's personal server that can be injected into any website's html files. In this situation, an attacker embeds a session-hijacking script within a website's forum input field, and the website accepts the field and posts it on their webpage. A victim browses to the page, which causes the website's html as well as the attacker's scripted to become rendered and executed in their browser. The victim's session information is sent to the attacker's server.

FIGURE 9: XSS ATTACK



In order to prevent XSS, all html input fields should be filtered to ensure html and scripts are not embedded within the string. In addition, web application front-ends should contain code and functionality that control's the website's visual interface. All data should be handled, validated, and pushed towards backend processes, preventing the browser from having access.

Just as web browser executable code can be easily manipulated, URLs can be used to access web application files stored on a server, giving a malicious user a gateway to backend code. This is easily preventable by creating a Whitelist and granting the user access to only the frontend code, throwing a 404 error whenever the user ventures outside of their allotted domain.

5.3 COMMERCIAL, OFF-THE-SHELF; FREE AND OPEN-SOURCE SOFTWARE; AND RE-USE SECURITY CONCERNS

Developers tend not to analyze or consider the security risks that Commercial, Off the Shelf (COTS), Free and Open-Source Software (FOSS), and re-use software impose on mission software systems. This software can produce weaknesses as readily as new development code.

As a part of FOSS culture, most contributions are done voluntarily, or with limited cost, by many different developers that may or may not interface with one another. This could indicate that no development process or procedure was followed, such as testing, peer review, etc. Open-source technologies also have their code bases exposed to the public, rendering itself an open slate to have potential vulnerabilities found with no grey area left on its functionality, interfaces, and design. Similar with FOSS, COTS software has unknown software pedigree, and its development standards and maintenance procedures are not able to be easily validated. The criticality within the introduced software system may be greater than the COTS software's original intent and design, leaving the new system incompliant to security standards.

Re-use and legacy code could use outdated packages and libraries with later found vulnerabilities, which could cause the software system to be at risk. During development and production, the software could have been adhering to the security standards of its era, but as time progresses, more vulnerabilities are discovered.

With these three categories of software, it is necessary to validate if they adhere to basic or corporate security standards. If not, re-use software and FOSS should be modified and adapted. One way to determine if software adheres to standards is to conduct static analysis and dynamic testing on the code (refer to section 4). The software should also be checked to certify if its maintained, otherwise, maintenance procedures should be adopted, when possible (refer to section 5).

Using an automated tool to manage FOSS can help determine if it is maintained and notify when a vulnerability is discovered. Tools that perform open-source dependency checks, also sometimes called software component analysis (a.k.a. software composition analysis or origin analysis) can help to ensure a project understands all of the open-source packages it is using, the versions of those packages, and reported vulnerabilities with those packages. Software developers should keep all of their included open-source packages in a single location which makes it easy to see what packages are in use in the project. Many open-source projects provide a cryptographic hash which can be used to ensure the downloaded version has not been tampered with. A basic check for programs should be to regularly reference the National Vulnerability Database (NVD) for all of the software packages, open source and commercial, in use (National Vulnerability Database, n.d.). The NVD assigns a Common Platform Enumeration (CPE) for each software package and version and allows users to search for all reported vulnerabilities for each CPE. The NVD assigns criticality ratings to each vulnerability that programs can use to assess the risk to their project. For larger software development efforts, a software component analysis tool may be helpful.

Supply chain source rise is worth considering within space systems due to the high value of assets in those systems. FOSS packages must be obtained from trusted sources/servers, but FOSS projects sometimes include thousands of developers from multiple countries. Missions should consider the

risk of vulnerabilities intentionally inserted into software they use; there have been reported cases of potentially malicious code being inserted into FOSS software repositories (Goodin, 2017).

5.4 Software Bill of Materials

The identification of FOSS or third-party code is becoming more important and more standardized. A "Software Bill of Materials" (SBOM) is effectively a nested inventory, a list of ingredients that make up software components. As stated by National Telecommunications and Information Administration (NTIA), "Most software depends on third-party components (libraries, executables, or source code), but there is very little visibility into this software supply chain. It is common for software to contain numerous third-party components that have not been sufficiently identified or recorded. Software vulnerabilities are both the byproduct of the human process of developing software and the increasingly frequent target of attacks into the software supply chain. If users don't know what components are in their software, then they don't know when they need to patch. They have no way to know if their software is potentially vulnerable to an exploit due to an included component – or even know if their software contains a component that comes directly from a malicious actor. The reality is this: when a new risk is discovered, very few organizations can quickly and easily answer simple, critical questions such as: "Are we potentially affected?" and "Where is this piece of software used?" This lack of systemic transparency into the composition of software across the entire digital economy contributes substantially to cybersecurity risks as well as the costs of development, procurement, and maintenance." (NTIA, n.d.)

SBOM operates under a standard pillar of cybersecurity, identification of assets (i.e., inventory) but it extends the idea to the minute level of detail similar to the nutritional facts on food products. You want to know all the ingredients to include breaking down items often referred to as "proprietary blend". From a cybersecurity perspective, security by obscurity has never truly sufficed therefore gaining the insight into every aspect of the software composition is critical moving forward.

Figure 10: Nutritional Facts

SUPPLEMENT FACTS
Serving Size: 11g (1 Packet)
Servings Per Container: 8

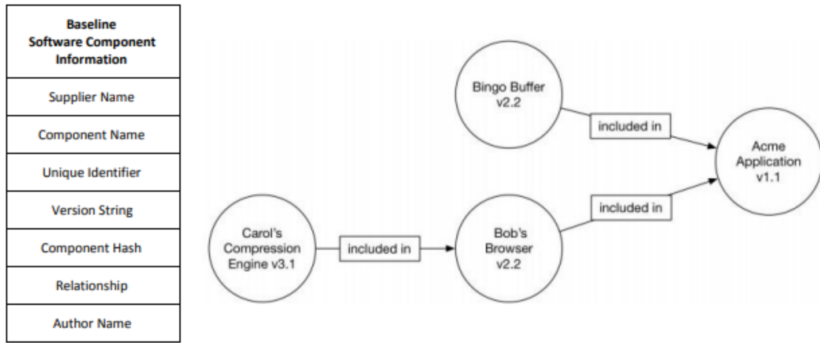
Amount Per Serving		% Daily Value
Total Calories	40	
Calories from Fat	1	
Cholesterol	0	0%
Sodium	4mg	0%
Total Carbohydrates	9g	4%
Dietary Fiber	1g	5%
Sugars	.5g	–
Protein	.5g	
Proprietary Blend	6.3g	*
Chicory Root Extract, Moringa oleifera Blend (Leaf Powder, Seed Cake, Fruit Powder), FoTi Root Extract 12:1		
Stevia	88mg	*

Percent Daily Value is based on a 2,000 calorie diet *Daily Value not established

INGREDIENTS: Proprietary Blend, Natural Tropical Flavor, Fruit Pe[illegible], Agar, Citric Acid, Stevia, Silica, Stearic Acid.

SBOM is undergoing a standardization process where the level of detail and contents of the metadata are being decided. There are currently three formats available which are variations of XML: SWID, SPDX, CycloneDX. It is likely one format will ultimately become the gold standard. The metadata/fields and format are important as that enables automation, scalability, and integration. NTIA published baseline SBOM that includes components in their assembled relationship. Each component has enough information to "uniquely and unambiguously identify" it (left), and the relationship of what upstream or child components are "included in" downstream or parent components (right) as depicted in the below figure.

Figure 11: NTIA Baseline SBOM



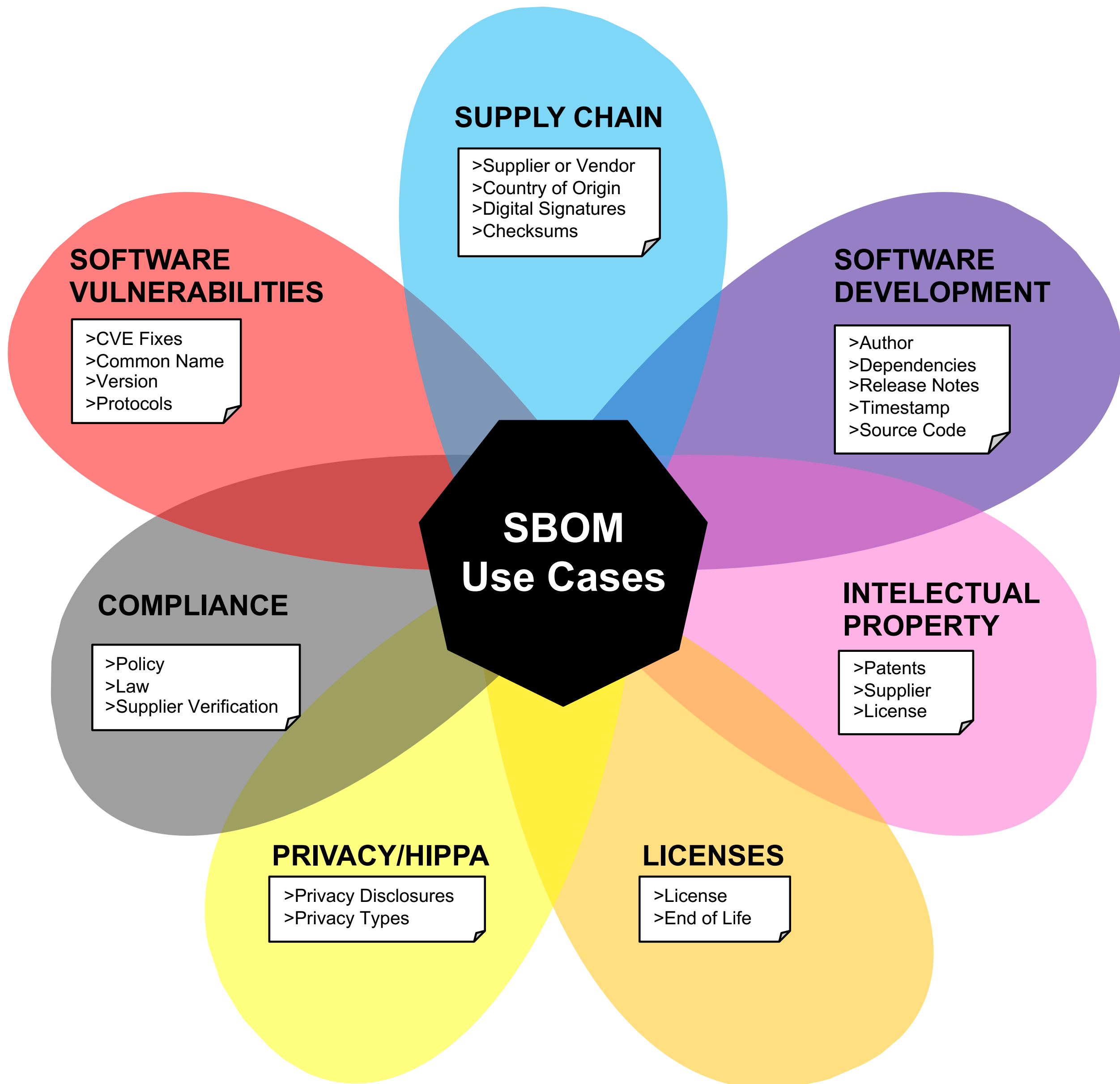
One example of an entity taking NTIA guidance and starting to standardize the format and fields is Microsoft who adopted the SPDX format as their standard as depicted below.

Figure 12: Microsoft Published SPDX Fields

NTIA field	NTIA description	SPDX 2.2.1 field
Supplier Name	The name of an entity that creates, defines, and identifies components	Package Supplier
Component Name	Designation assigned to a unit of software defined by the original supplier	Package Name
Version of the Component	Identifier used by the supplier to specify a change in software from a previously identified version	Package Version
Other Unique Identifiers	Other identifiers that are used to identify a component, or serve as a look-up key for relevant databases	Package SPDX Identifier
Dependency Relationship	Characterizing the relationship that an upstream component X is included in software Y	Relationship
Author of SBOM Data	The name of the entity that creates the SBOM data for this component	Creator
Timestamp	Record of the date and time of the SBOM data assembly	Created

SBOM has multiple use cases and as the approach and technology progresses additional use cases could be discovered. The below graphic provides a snapshot in time on today's known use cases for SBOM. All of these would apply to space systems.

Figure 13: SBOM Use Cases



5.5 Compilation Guidelines

In compiled languages, most compilers offer various compile-time options in order to harden the final compiled binaries against potential attack vectors. Even if the code contains a potential attack vector, certain compiler options will help prevent loading of arbitrary code or libraries to help limit the damage caused. These options are often platform and language dependent, so portability must be considered for cross-platform projects, even when using the same toolchain even.

Due to the extra resources consumed to perform runtime checks for the various forms of protection offered, a consideration to take when using these compilation flags is that the features designed to harden the compiled code may cause measurable performance degradation.

The GNU Compiler Collection (GCC) provides various compiler flags to help mitigate potential memory-based attacks on compiled code. The most common flags are as follows:

```
GCC: -fstack-protector
Visual Studio: /GS
Clang: -fsanitize=<option>
```

The stack protector keeps an application's stack in check and causes the program to self-terminate if any stack tampering is detected. There are multiple levels of this flag for the GCC variant, as appending "-all" to the end of the above flag will wrap stack protection around all possible functions, even if not strictly necessary. There is also a "-strong" variant of the flag that balances the stack protection of the compiler with speed.

```
-fstack-clash-protection
```

This GCC flag attempts to prevent stack clashing attacks by only allocating a limited amount of memory at a time.

```
-D_FORTIFY_SOURCE=#
```

This option will fortify any compiled code against buffer overflows when using certain C memory functions. There are two levels of checks that the flag can be set to, 1 or 2, where 1 consists of the safer checks that are guaranteed to not disrupt the program's flow. Level 2 of this flag runs more checks but may affect how some programs run.

For Microsoft's Visual Studio compilers, the stack protector is enabled by default. With Clang, the AddressSanitizer code instrumentation suite is built into the compiler, allowing the flag shown above (when used with the options "address" or "bounds").

5.6 Authentication and Password Management

Space mission software often relies on authentication mechanisms in order to limit access to authorized individuals. Best practice is to use multi-factor authentication or two-factor authentication whenever possible. Multifactor authentication does not completely secure a system, but it slows down an adversary by making it significantly harder for them to gain initial access to a system. When password authentication must be used by itself, the systems should enforce password complexity, length, expiration, and history requirements. In practice, these requirements may not be sufficient. It is easy for users to create passwords which follow keyboard patterns that meet the complexity requirements but are easy to guess for adversaries that know common

patterns. The purpose of enforcing password complexity and length requirements is to make it difficult for adversaries to find passwords by brute-force guessing if they obtain a set of password hashes via a system breach. Obtaining the cleartext passwords often allows them to escalate their initial access to other parts of the system or administrator access. Microsoft explains why long passwords are more secure than complex passwords in a blog post (mstfcam, 2015). Long passwords may be easier for users to remember too: they can use phrases of English words that are easy to remember but are still difficult for adversaries to brute-force guess. Enforcing password expiration and history is important too because if an adversary obtains a password, its useful lifetime should be short.

6 Analysis Tools and Techniques

6.1 Static Analysis

Static analysis is the analysis of software without executing the code. This can range from essentially fully building the software to merely parsing the code "fuzzily" without the need of any dependencies or build tools. Most static analysis tools provide insights into the codebase through additional metrics and attempt to detect many kinds of issues with the code, including many of the aforementioned CWEs. A well-known, fundamental introduction to the techniques of static analysis can be found in (Allen & Cocke, 1976).

Static analysis can be useful in situations where the build environment of the software is not available to a reviewer, or to enforce coding standards that are not necessarily enforced by a compiler (such as organizational or stylistic choices). Static analysis is very useful early in development when the size of the codebase is small – the analysis will be quicker, and it is easier to establish a foundation for continued analysis. Many tools can have subsequent analyses automatically ignore known false positives and are also able to track changes in the amount of detected vulnerabilities over different builds. Just like managing an inbox, it is usually easier to deal with findings as they crop up early in development instead of deal with them all at once at the end of a project's development cycle. With a baseline of stylistic coherence and coding practice enforced by static analysis, any quality assurance processes such as code review can focus on the detection of more subtle errors.

Static analysis cannot know anything about the software that is determined at runtime. This makes static analysis less capable of reliably detecting classes of issues such as memory leaks, incorrect calculations, and divisions by zero. Many of these issues are detectable statically but require the use of computationally expensive methods such as abstract interpretation1. Tools that can do so are often highly specialized and more expensive than generalist ones.
Static analysis is unfortunately not a magical automated safety net against human error. Due to the comparative lack of context compared to dynamic testing, static analysis is very prone to false positive findings. Depending on a given tool's approach to the analysis, false positives can comprise an overwhelming majority of the results, making it necessary to have an engineer review

[1] Abstract Interpretation is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems (Cousout, 2005). It is often used to prove or certify software against certain kinds of runtime errors.

the results and separate the real findings from the chaff - a potentially extremely time-consuming task depending on the rigor of the tool used and the size of the codebase. Despite the propensity of static analysis to produce false positives, it is also still possible for some security vulnerabilities to pass through undetected.

6.1.1 Tool Choice

Perhaps the most important factor in static analysis tool choice is compatibility with the system's languages. While some static analysis tools are polyglottic, others are specialized to work with only one or a few given languages. A combination of tools may be necessary to cover all the components of the software.

An important consideration in choosing a static analysis tool should be the class of issues that impact the software the most. For example, a system that will often handle untrusted input (such as a form entry web application) will find it useful to focus on detecting Data Processing Errors (CWE-19), many of which concern the mishandling of untrusted input which can lead to code injection vulnerabilities. A system with a defined set of inputs would most likely not benefit from such scans and might instead benefit more from focusing on logic errors.

Some tools are better at covering certain CWEs than others. As part of building a development/assurance pipeline it will require a combination of complementary tools. The tools listed below addressed a high percentage of the most critical CWEs mentioned in Section 5.2 and Appendix A; however, this list is merely a snapshot in time and a non-exhaustive list. The listing of these tools does not imply endorsement as the list is for informational purposes only.

Table 4 CWE Coverage of Static Analysis Tools

Programming Language and System	Combination of Tools to Identify CWEs
C/C++ Ground Systems	CodeSonar Fortify Parasoft C/C++ Klocwork CheckMarx Coverity
Java Ground Systems	Grammatech CodeSonar Fortify Parasoft Jtest Klocwork Spotbugs CheckMarx Coverity
C/C++ Flight Systems	Fortify CodeSonar Klocwork Checkmarx Coverity

6.1.2 Scanning

Static analysis can be time consuming, so it is important with large codebases to only scan code that pertains to the system proper and has the potential to raise new findings, such as code that is freshly written or old code that interacts with new code. Test code usually need not be scanned at all.

The frequency of scanning depends on how quickly scans can be performed. For something as basic as a style check, it is doable every time a developer commits and pushes code. For more in-depth analyses that take on the order of minutes or hours, it might make more sense to integrate it into a nightly build cycle or other continuous integration toolchain. Best practice is to have the analysis occurring nightly or on every commit/merge into the baseline.

The static analysis tool should be configured as appropriately as possible. If present, the tool should have access to the same libraries the code will in production. The tool should be configured to use the same compiler standards as the target. Some static analyzers can still perform a limited analysis with missing libraries, but it is recommended that the environment the static analyzer work with be as close to production as possible. Many C/C++ codebases are required to be compiled to even do the static code analysis, therefore integrating into the build process is key.

While static analysis can generate worthwhile findings about a codebase, it is important to adjudicate the findings efficiently and have a consistent strategy for dealing with fixes or mitigations. Every finding should undergo a risk assessment to determine what course of action should be taken to deal with it.

Whether or not a static analysis finding is a false positive is the first matter of concern. Only when this has been determined by a reviewer can its criticality be determined. The criticality of a finding is the next most important factor to consider. Criticality can be considered a combination of the likelihood of the bug being triggered in production and the effects of the bug on the goal of the mission. Depending on the stage of development at which the finding arises, it can be prohibitively costly to implement a fix or workaround. However, if the bug is unlikely enough, benign enough, or both, then it may not pose a great enough risk to the mission to warrant spending more budget fixing it.

It is useful to classify vulnerabilities according to a defined set of criteria when determining their criticality and what action to take. There are several criticality scales and each tool has their own criticality scale. There are scales like OWASP Top 10, SANS Top 25, as well as operational discrepancy categories which have be adapted for a software context (U.S. Air Force, 2015). For example, the discrepancies or defects are organized by Category (1 or 2) and Priority (Emergency, Urgent, or Routine). Category 1 discrepancies have no acceptable workaround, while category 2 discrepancies do. Emergency defects are generally most severe and involve life-critical and/or safety-critical weaknesses, while Urgent defects affect operational capability to a lesser degree. Routine defects are merely inconvenient and involve some compensation by the users of the software.

Using predefined criticality scales typically do not consider operational context of the specific mission. Therefore, it is recommended to develop a mission specific prioritization. One method of

performing this prioritization is leveraging CWEs and performing a mission specific prioritization using something like common weakness scoring system (CWSS). The CWSS is an ongoing effort by MITRE to provide a mechanism for prioritizing software weaknesses in a consistent manner. A modified version of CWSS was used to prioritize CWEs for both ground and spacecraft listed in Appendix A. The list in Appendix A was curated considering the nuances of the operational environment for ground systems and well as spacecraft. Developers tend to rely on the weakness ranking subjectively outputted by MITRE or the various scanning tools used to check code. The problem is these tools do not have the necessary knowledge of space systems to determine which weaknesses are more or less important to mission. Appendix A can be used as reference, but ultimately the mission owner should perform their own prioritization given their mission context and risk tolerance. The list in Appendix A is in order of priority from highest to lowest, but all the listed 386 CWEs are considered high priority (i.e., priority one CWEs) for space systems but they are listed in descending priority order.

When addressing potential software vulnerabilities in a system, it is important to follow a defined process that combines the following:

1. Organized risk assessment (during which static analysis generates findings)
2. Communication and feedback with the engineering team (where they may be made aware of or clarify any findings, such as additional context on possible false positives)
3. Addressing residual risks and mitigation plans after changes are implemented

This is just one example of a review process incorporating static analysis. It sets up a good feedback loop in the context of static analysis whereby the findings are treated appropriately and systematically and can be used to establish a flow for tracking metrics on vulnerabilities and false positives detected over time. Most importantly, it ensures that findings are worked through with due consideration and minimizes the number of vulnerabilities that slip through.

6.1.3 Third Party Code Analysis

It is highly recommended the developers perform their own static analysis using automated tools looking for coding standard deviation and code weaknesses/vulnerabilities. However, third party reviews often take place as well. These should be performed in tight coordination with the development organization. Third part reviews should aspire to utilized different analysis tools than the development organization, in addition to peer reviewing/validating the developer's usage of automated tools. Oversight is often needed as developers may accept risk or elect to not resolve a defect that could lead to mission impact.

For third party reviews, it is important to consider the developer's possible reasoning behind writing software a certain way when examining a static analysis finding. Even an ostensibly serious finding such as the use of functions vulnerable to buffer overflow needs to be contextualized – the developer may have taken steps to manage the risk involved in using those functions to meet a performance requirement, for example. Communication with the originating developers is very important in ensuring that third party reviewers know the standards to which the developers work.

6.1.4 Analysis Without Source Code

Scanning source code or performing manual review is required in order to identify findings that may be exploited and expose the system, placing systems and data at risk. There are many types of scans that detect potential vulnerabilities, among them static code analysis is the method that provides a more comprehensive and thorough look at the risk posture of software. Access to the source code is required when static code analysis is performed to identify the potential vulnerabilities and to examine them after the scan to determine exploitability and risk. Sometimes the users of the software do not have access to the source code, and they do not have a clear picture on vulnerabilities that exist in the code when they use the software.

Missions often purchase commercial software with no data rights due to budget constraints or other contractual considerations. In the era where cyber threats only become more sophisticated this presents a security loophole that can have catastrophic consequences, leaving the mission without visibility of the software's assurance posture. Lack of data rights for newly acquired systems is only part of the problem, access to code is also not be possible on legacy systems.

There are tools that perform binary analysis on installers and executable files where we do not have access to the source code. The problem with these tools is that the technology is immature and usually the results are vulnerabilities in the assembly language. To evaluate assembly language vulnerabilities requires specialized skills and knowledge that is not widely available. Even with knowledge available it takes time to assess the vulnerabilities, preventing these methods from scaling to the large amount of COTS software used. The reverse engineering required for these approaches may also be illegal for some software due to provisions within the terms of service or software license.

As a result, mission owners are left without many options regarding COTS software vulnerability analysis, and there are no standards and guidance on the process that needs to be followed to provide the assurance that the software is safe to purchase and use. The options available currently to find potential vulnerabilities in software that do not have source code are to:

- Perform binary analysis on custom code: As discussed above specialized skills and reverse engineering are required to be able to assess the vulnerabilities. This type of analysis is immature and generates a lot of false positive results, but it does possess the ability to discover zero-days or unknown-unknowns. In addition, it covers only a fraction of the vulnerabilities that may exist in the source code and critical findings may be missed resulting in increased security risk.
- Perform Software Composition Analysis: This is a more mature technology and can successfully identify third party components residing in the code base. The technology will list any CVEs associated with the components and will provide a lot of other useful information such as applicable software licenses. Composition analysis may identify third party libraries and some commercial software used including their version and existing well know vulnerabilities associated with the version. This method does not identify the vulnerabilities in custom code that is not provided by a third party, or in cases where the third-party library is not recognized by the composition analysis tool(s) being used. This method of analysis can only detect the known-knowns within the software vice zero-days.
- If the user of the software does not have access to the code to check for potential vulnerabilities and does not have the ability to perform either binary of software

composition analysis, then the decision is made to accept or not the software with the unknown risk related to security posture of the software.

6.1.4.1 BINARY ANALYSIS

Binary code analysis, also referred to as binary analysis, is threat assessment and vulnerability testing at the binary code level. This analysis analyzes the raw binaries that compose a complete application, which is especially helpful when there isn't access to the source code. Because a binary code analysis evaluates stripped binary code, software can be audited without vendor or coder cooperation. It can also be used to analyze third-party libraries, allowing a richer analysis and better visibility into how applications will interact with libraries.

Binary analysis requires specialized skills to understand the results that are usually in assembly language. It is a time-consuming effort, and the results of the effort are hard to translate into a high-level language to pinpoint where in the code the vulnerability resides. The reverse engineering that is required to understand the results may be illegal depending on the data rights related to the software that is examined. This method is like standard static code analysis, there is expertise necessary to effectively perform binary analysis. Various tools have differing levels of skill to deploy and execute against binaries. There is no "one tool" or "set it and forget it" when performing binary analysis. Interpreting the results takes knowledge of software development, assembly, various hardware architecture platforms (ARM, PPC, MIPS, SPARC, x86_64) as well as various operating systems (VxWorks, RTEMS, Linux, Windows).

If vulnerabilities are found during binary analysis the vulnerability report can be presented to the owner of the software with the request to fix them before software acceptance. Areas of consideration when performing binary analysis are the software operating environment and file types available; these will determine the tools used and the analysis performed.

6.1.4.2 SOFTWARE COMPOSITION ANALYSIS

Component Analysis is the process of identifying potential areas of risk from the use of third-party and open-source software and hardware components. Component Analysis is a function within an overall Cyber Supply Chain Risk Management (C-SCRM) framework. A software-only subset of Component Analysis with limited scope is commonly referred to as software composition analysis.

Software Composition analysis is a mature technology that provides valuable information related to the open-source components and third-party libraries that are found in the software. The information usually includes CVEs, software version, license types etc.

6.1.4.3 MALWARE ANALYSIS

Modern antivirus software can protect users from: malicious browser helper objects (BHOs), browser hijackers, ransomware, keyloggers, backdoors, rootkits, trojan horses, worms, malicious

LSPs, dialers, fraud tools, adware, and spyware. Some products also include protection from other computer threats, such as infected and malicious URLs, spam, scam and phishing attacks, online identity (privacy), online banking attacks, social engineering techniques, advanced persistent threat (APT) and botnet DDoS attacks.

In addition to signature-based malware detection methods, there are sandboxing environment that are freely available to enable a more behavioral analytics approach. Software solutions like Cuckoo Sandbox (Cuckoo) and Assembly Line (ASL) can provide this sandboxing environment. If the software is not sensitive, VirusTotal (VT) provides a mechanism to submit binaries for free which analyses the software using over 70 virus scanning solutions at once.

6.2 DYNAMIC TESTING

Dynamic testing has proven valuable in industry for finding defects and security weaknesses in code. Microsoft uses an internally-developed dynamic testing tool called SAGE to test Windows and Office software. Microsoft reported SAGE had consumed 100+ computer-years and found hundreds of bugs (Godefroid, From Blackbox Fuzzing to Whitebox Fuzzing towards Verification, 2010). Microsoft noted that SAGE typically runs last in their testing pipeline, meaning it found bugs that all of the other testing methodologies such as static analysis had missed (Godefroid, Levin, & Molnar, SAGE: Whitebox Fuzzing for Security Testing, 2012). Stanford developed a tool called KLEE which they used to test 452 programs from the core Unix utilities and found 56 serious bugs (Cadar, Dunbar, & Engler, 2008). Dynamic testing was used extensively at the DARPA Cyber Grand Challenge in 2016: at least two of the top teams, ForAllSecure and Shellphish, used a combination of dynamic testing techniques (Defense Advanced Research Projects Agency, n.d.), (Avgerinos, et al., 2018), (Shoshitaishvili, et al., 2016).

Dynamic testing discovers flaws that appear at runtime by running a compiled binary with unusual inputs. It attempts to cause undesired effects such as crashes or hangs by inducing conditions that were not anticipated by the developers. Dynamic testing complements static code analysis: dynamic testing tends to find certain types of flaws more readily than static analysis. Two good examples are buffer overflows and use-after-free weaknesses. Static analysis tools have a difficult time determining the size of buffers that are allocated at runtime, and so they do not always correctly flag buffer overflows. They also have a difficult time tracing complex execution paths between memory allocation and corresponding memory free operations, and so they often cannot correctly flag cases where the software will use a pointer after the memory has been freed (use-after-free). Dynamic testing can discover these flaws at runtime by executing the code and looking for memory errors that indicate one of these defects.

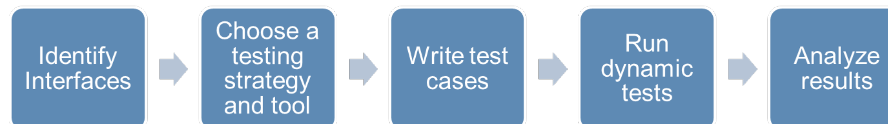
Dynamic Analysis encompasses a variety of techniques:

- Manual or semi-automated testing (i.e., penetration testing) using an execution environment (i.e., high-fidelity simulator or with hardware-in-the-loop environments).
  - In addition to traditional penetration testing techniques, automated vulnerability scanning would reside within this technique
- Traffic analysis: while not technically dynamically executing the software under test, traffic analysis (i.e., protocol analysis or radio frequency analysis) requires the software to

be executing so the tester can analyze the incoming and outgoing traffic. This is common approach for intrusion detection systems on a traditional network where it analyzes the packets on the network
- Symbolic execution: use theorem provers to find weaknesses in binary code
- Sanitizers: instrument code to show violated conditions (e.g., buffer overflows)
- Fuzzing: send random inputs to a program to see if they crash the application

A widely used type of dynamic testing is called fuzzing. Fuzzing generates random inputs for the program to bypass input validation or programmer assumptions and cause the program to crash. For example, if the programmer assumed input would always be alphanumeric characters, fuzzing may cause a crash by sending a character outside of those bounds. Symbolic or concolic execution builds a representation of program execution as a series of logical statements, then uses a theorem prover to decide whether certain security violations may occur. For example, a symbolic analyzer would start with a symbolic variable representing user input. It would then step through a program one CPU instruction at a time. At each instruction, it would add a logical statement representing that instruction's operation on user input (i.e., addition, subtraction, logical operators). When it reaches an operation that could be a potential security weakness such as a possible buffer overflow, the theorem prover decides whether any possible input could result in a lookup outside the bounds of that array. Sanitizers instrument code to check security or various memory or thread safety conditions throughout execution. When they reach a point where the instrumentation detects a violation, they report that violation. For example, a memory allocation sanitizer would instrument all memory allocation and frees and report any memory leaks when the program terminates.
A general diagram of the dynamic testing process is:

FIGURE 14: EXAMPLE TESTING PROCESS



The following sections describe each of those steps and useful tools in detail.

6.2.1 PREPARING FOR DYNAMIC TESTING

Dynamic testing typically requires more preparation and analyst effort than static analysis. This is because dynamic testing for most applications depends on the interfaces and structure of the application. Preparation should involve identifying the interfaces to test, choosing a testing strategy and tools, and writing test cases.

6.2.1.1 IDENTIFY INTERFACES

The first step in dynamic testing is to identify what interfaces to test. Dynamic testing should focus on interfaces that that accept input from users or across trust boundaries. Trust boundaries are interfaces to external or potentially untrusted sources of data. Some examples are file inputs from external systems, input fields in graphical user interfaces or web frontends, scripting engines, and network services. There are additional reasons to perform dynamic testing than discovering

security weaknesses: dynamic testing can discover bugs that would result in system crashes, potentially affecting the reliability of critical systems.
There are many examples of defects or security weaknesses that dynamic testing has found in code. File inputs or parsers introduce potential weak points in code. For example, fuzzing discovered that malformed image files can cause crashes and possibly security issues by overflowing heap memory in the ImageMagick image processing library. (National Vulnerability Database, 2017) Fuzzing identified incomplete fixes to the Shellshock vulnerability reported in bash. (National Vulnerability Database, 2014)

6.2.1.2 Choose a Testing Strategy and Tool

There are different strategies and associated tools for dynamic testing. White box fuzzing instruments the code to measure how many branch paths are exercised. This requires compiling the source code with instrumentation in place. Black box fuzzing uses an unmodified binary but does not measure testing coverage. White box fuzzing tends to exercise many more branches through the code, and thus finds more defects. White box fuzzing is not always an option, however, especially in third party reviews, when testing legacy code without source, or when preparation time is limited.

A mutational fuzzer, sometimes called a "dumb" fuzzer, modifies the input randomly without considering any knowledge of the underlying data structure. A generational or "smart" fuzzer uses a specification of the underlying data structure to make "smart" decisions about how to randomize the input. For example, a generational fuzzer may calculate CRC or checksum values where appropriate, whereas a mutational fuzzer would have to randomly find the correct value. Generational fuzzing should be used with care, however: a major reason to perform fuzzing is to test how a program reacts to unanticipated inputs, and the random inputs of a mutational fuzzer overcome any biases of the developers or testers.

Microsoft studied different fuzzing strategies and found that a combination of white box and generational fuzzing discovered the most defects, but also required the most effort. (Neystadt, 2008) Programs will need to determine the best strategy given risk tolerance and resources available.

If fuzzing is not the dynamic testing strategy and the software is running on an embedded processor (e.g., PowerPC) then either a hardware in the loop or a high-fidelity simulator using an instruction set simulator could be used as the method the execute the test cases.

6.2.1.3 Writing Test Cases

Dynamic testing requires test cases to exercise the code and find defects. Ideally, these test cases should target the identified interfaces and bypass as much other code as possible. For example, network servers often go through a process of loading configurations and starting services. If the identified interface is a single network listener, then a test case would include realistic configuration values but bypass as much of the server startup process as possible. If the code under test followed a unit test development methodology, those unit tests are often candidates for dynamic testing. For large monolithic software projects, however, creating test cases may mean extensive modifications to the source code if the project does not already have a thorough set of unit test cases.

There is sometimes a tradeoff between the amount of effort required to create test cases and how much testing can be performed: a large software package may be left as-is, minimizing the upfront work to prepare for dynamic testing, but dynamic tests will run slowly and will be less likely to discover issues because fewer tests can be run.

Fuzzing tools and frameworks often use standard input to send random data to the program, and so test cases should accept input from standard input and pass it directly to the code under test. For example, a test case for a network service should bypass the normal mechanism to accept data over the network and accept that data from standard input instead. There are tools that can help with this process: for example, a tool called preeny automatically simulates network traffic by taking data from standard input (zardus, 2018).

Some types of dynamic testing such as fuzzing require a seed input to start the randomization process. The seed could be a one-pixel image file for an image processing program, for example. The input seed speeds up the testing process by randomizing inputs from a valid starting structure. In the image example, without a seed, the fuzzer would have to randomly stumble on the structure of image files.

6.2.2 Fuzzing Tools and Frameworks

American Fuzzy Lop or afl-fuzz is a widely used and effective fuzzing tool for dynamic testing. It is developed by a Google security engineer and includes some sophisticated features to find as many defects as possible. If source code is available, afl-fuzz can instrument the binary during the compilation process to measure branch coverage. afl-fuzz uses a sophisticated mutational fuzzing approach. The fuzzer uses an algorithm resembling a genetic algorithm to find inputs that exercise as much of the code as possible. This is an improvement over purely random fuzzing because it will use inputs that exercise new execution paths to discover even more execution paths. afl-fuzz works on Linux with the gcc or llvm compilers. It expects test cases that receive input via standard input or via a file argument on the command line. It also requires some seed inputs to help start the fuzzing process. afl-fuzz can use a dictionary to help guide its inputs which speeds up exploration when the program under test expects certain key words (lcamtuf, 2017).

Carnegie Mellon CERT provides another fuzzing framework called the CERT Basic Fuzzing Framework (BFF). BFF is a mutational fuzzer and provides many of the same features as afl-fuzz. It also provides some additional tools for call trace analysis and triaging findings (Householder, 2018).

6.2.3 Symbolic and Concolic Execution

Symbolic and concolic execution tools are relatively new but are promising for finding runtime defects that fuzzing does not find. For example, fuzzing has trouble finding corner cases in the execution path that are highly unlikely using purely random input. These tools were used at the DARPA Cyber Grand Challenge by several of the top teams including the winner, ForAllSecure, who deployed their tool Mayhem. Symbolic execution runs software binaries through a CPU simulator which builds a symbolic representation (as a set of logical statements) of input and calculations as it executes. When the program reaches a point of interest, such as an index into a buffer, the symbolic analyzer uses a theorem prover to solve the symbolic representation and determine if any input could lead to security violations such as an index beyond the bounds of the

buffer. This is an extremely powerful method for analyzing software and can find rare corner cases that fuzzing misses with its random approach. Symbolic execution suffers from exponential path explosion, however. Each time the program branches such as at an if statement or loop, the symbolic analyzer must fork the symbolic representation and follow both paths. For analysis of a few thousand lines of code, this is feasible using current computing technology, but for complex programs it quickly becomes computationally intractable. Concolic execution, a mix of concrete and symbolic, helps speed up the process by executing some of the code directly without building a symbolic representation. Segments of the code which do not need to be analyzed such as initial configuration or library functions may be executed concretely, and only the portions of the code which perform calculations on program input are executed symbolically.

There are several symbolic execution tools available for use in testing space ground software. ForAllSecure provides its Mayhem analysis as a service. (ForAllSecure, 2018) The University of California Santa Barbara provided their symbolic execution code called Angr as open source. (UC Santa Barbara, 2018). It is primarily intended for reverse engineering in cyber competitions but is also capable as a security defect analysis tool. It currently requires a significant amount of analyst effort to achieve results.

6.2.4 Memory and Thread Sanitization

Memory and thread analysis are valuable in a secure coding toolset, as they provide runtime-level detection of critical memory or thread-based errors that usually cannot be detected through static code analysis. While the following tools add overhead through adding their instrumentation frameworks at the binary level, the results these tools can provide are invaluable to find critical errors that can open serious attack vectors on a host system.

Valgrind is a software tool for detecting incorrect memory usage and leaks at various points throughout the lifecycle of an application; during runtime, through the final return or termination statement. Valgrind is composed of multiple modules that combined, provide the ability to validate the memory of the application throughout its execution. As a result, it can be a valuable tool in detecting memory misuse or leaks that cannot or are not detected by static code analysis, as unexpected memory misuse can occur without detection.

Valgrind is granular in its memory error detection, which allows the programmer to pinpoint the exact location in code the errors are occurring, when the code is compiled with debug symbols. The following code provides an example of Valgrind's memory error detection capabilities, as well as the corresponding Valgrind runtime output: (Valgrind Developers, n.d.)

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

Source: (Valgrind Developers, n.d.)

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40
alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Valgrind also provides many core features, such as integrating with the GNU debugger (gdb), as well as various modules for other types of runtime errors, like helgrind and DRD for thread errors or SGCheck for detecting overrun memory in stack and global arrays. In general, the most common module the programmer will use in following secure coding guidelines is Memcheck.

A suite of runtime sanitizers is available for compiled languages such as C or C++ through the LLVM project, which provides the Clang compiler.

Another robust tool for detecting memory errors or misuse at runtime is AddressSanitizer, which is a proven tool for detecting common memory corruption errors in the heap, stack, or global memory, such as buffer overflows or use-after-free. Both of these memory bugs are possibly critical vectors for exploit that should be patched, as they may allow an attacker to reach outside the function bounds of the application (Google, n.d.).

Additionally, AddressSanitizer has a complementary sanitizer component called LeakSanitizer that is able to detect memory leaks much like Valgrind's complete instrumentation suite. Although normally instrumented into binaries with AddressSanitizer, to reduce load, the LeakSanitizer can instead just be linked against the code, which provides basic leak detection functionality.

ThreadSanitizer is a compilation-time dynamic testing tool that detects data races, which occurs when two threads attempt to access the same variable and run multiple operations on said variable. As a simple example, consider the following C++ code:

```
int var;

void Thread1() {  // Runs in one thread.
  var++;
}
void Thread2() {  // Runs in another thread.
  var++;
}
```

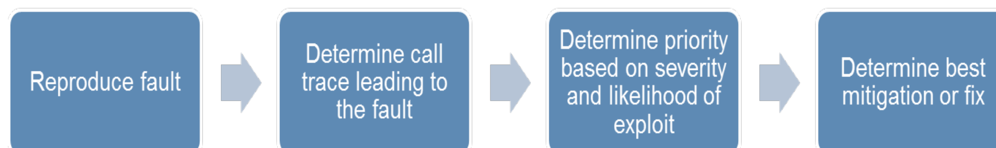
Source: (Google, n.d.)

In this code, the two separate threads will access the global variable var, and as a result, running the application will likely result in an unexpected value for var. In general, thread-based data races are likely to be more subtle or unintended, so the use of ThreadSanitizer can help maintain data integrity (also see the section on race conditions).

6.2.5 Analyzing and Mitigating Findings

Dynamic testing reveals inputs that cause undesirable effects such as crashing the program, but follow-up analysis is required to find where the program failed and where the most appropriate fix should be applied. A useful process for each finding is:

Figure 15: Analyzing Faults from Testing



Reproducing the fault is usually straightforward, although some faults may depend on some external state such as time or files. The first step is to ensure the fault is readily reproducible by the analyst.

The next step is to determine the call trace leading to the fault. This will pinpoint the location in the program that experienced a fault, and all the functions and intermediate arguments between input and the fault. Debuggers are useful tools at this stage. On Linux platforms, the free GNU Debugger is a useful tool. On Windows, Microsoft provides debugging tools as part of the Visual Studio package. An excellent option for security analysis is the IDA Pro debugger. Memory checkers such as Valgrind and AddressSanitizer are useful during this step in the analysis because they reveal details about memory violations that occurred. Binary analysis frameworks such as radare2 may be useful as well (radare2, 2018).

The next step is to determine the priority to fix or mitigate the defect. This depends on the severity and likelihood of someone exploiting the defect. Analysts should consider what data source could induce the fault and how difficult it would be to induce the fault. Severity also depends on effects the fault could cause. Many faults lead to crashes or hangs, which affect the reliability of the software, but may not be useful for security violations. Memory write violations that affect the stack or heap are more serious and should be investigated as potential security issues.

Finally, the analyst should determine the best mitigation or fix for the defect. The best fix is sometimes not the exact location where the fault occurred. If the fault resulted from invalid input, the best fix may be input validation early in the call trace. If the fault occurred in a library function, the best fix may be at the Application Programming Interface. Sometimes dynamic testing reveals several defects that are related by a single function call, in which case a fix in that function may solve many issues.

6.2.6 Testing Environments

Dynamic analysis can be performed using different types of environments:

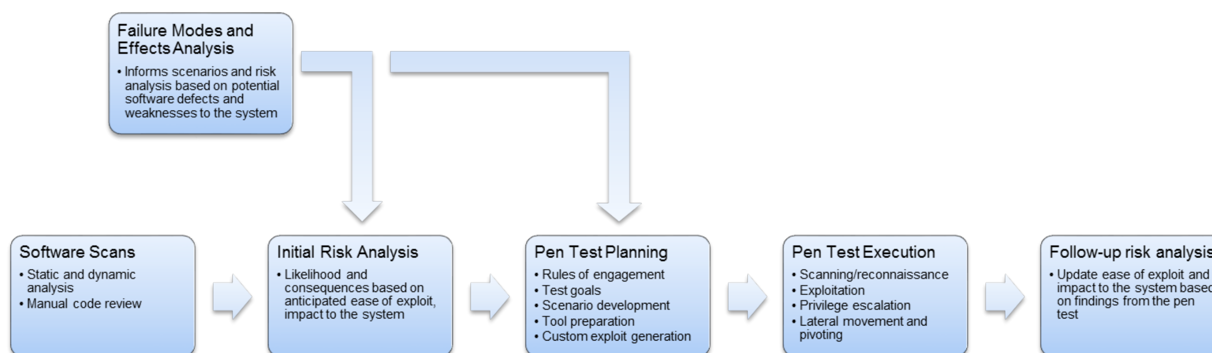
- Virtual, using instruction set simulators, virtual machines, or containers in a controlled environment. This maybe the cheapest and more scalable option that leaves a small footprint, but it requires virtualization expertise. The behavior of the system may not be exactly like the one in the physical environment. Snapshots, traffic captures and tool outputs are the evidence that can be used to assess the potential weaknesses.
- Physical, this will be the exact duplicate of hardware, firmware, and software. It will include analysis of the equipment used. Using the physical environment is not as scalable as using a virtual environment and it has limited use and there are size restrictions. To build the replica of physical environment space considerations are important as well as cost. Storage of hard drives may become problematic and availability to clean hard drives too. The assessment will use evidence of traffic captures and dynamic analysis tools.
- Hybrid, a combination of a virtual and physical environment to perform dynamic analysis. This approach is very common in forensics and it requires unique hardware, firmware, and software combinations. To build this approach a small lab is sufficient depending on unique HW/FW and the availability of clean hard drives for different events. The weaknesses with this approach can be assessed using the tools results, traffic captures and snapshots of the virtual environment.

6.2.7 Targeted Penetration Testing

Penetration testing can find defects in a fully integrated, configured, and deployed system. Typical penetration testing uses tools and techniques to discover common vulnerable software and configurations. This is valuable for traditional networks that use mostly commercial and commonly available open-source software but does not adequately address systems with a large amount of custom software. For space systems that include custom or domain-specific mission software, it is recommended to target penetration testing that attempts to discover and demonstrate defects in the mission software.

Penetration testing should be performed by a team that is independent of the developers. It can be adversarial, where the penetration testing team is given limited information about the system and may not work with the developers, or cooperative, where the penetration testing team works closely with the developers to understand the system and discover its weaknesses. Adversarial testing is useful for understanding a system's weaknesses to outside parties, but cooperative testing is more useful for discovering as many weaknesses as possible.
The following diagram shows a target penetration testing process:

FIGURE 16: EXAMPLE PENETRATION TESTING PROCESS



Targeted penetration testing should use results from static and dynamic software analysis. Often the developers will rate an issue as low risk due to assumptions about how the software will be used, but penetration testers may find conditions in which that issue is higher risk than anticipated. If the system has performed a failure modes and effects analysis, those results should be used to highlight critical portions of the system. Both activities should come with an initial risk analysis for each finding. Penetration test planning should include rules of engagement such as portions of the system in scope, allowed tools and actions, and contact information for parties involved. The goals of the test should be discussed and agreed upon with system owners. For example, if the system owner is concerned about threat scenarios, those should be discussed and included in the test. High-level scenarios ensure the test will be realistic and produce useful results. For example, insider threat or network attack are two scenarios that may be used. Tools such as network scanners and scripting languages should be prepared. Finally, custom exploits may need to be developed to target weaknesses in the software.

Penetration test execution follows a traditional model of scan, exploit, escalate, pivot, and repeat. There are many resources available to help train for and conduct a penetration test. The SANS Institute and Offensive Security both offer highly valuable training (SANS Institute, 2018), (Offensive Security, 2018).

Finally, a follow-up risk analysis should be performed. This involves re-evaluating the initial risks given the findings of the penetration test. The results should be discussed with the developers and a plan for mitigating or fixing findings should be developed and implemented.

7 Software Acquisition Security Best Practices

When acquiring software, it is key to ensure confidence in a system, that the system functions as intended, and that the system is free of vulnerabilities. Confidence regarding contractors' assurance activities comes from obtaining appropriate information that a program office and others can understand and that supports the claims about the functionality of the software as well as the addressing of exploitable constructs in the system. The use of standardized collections of weaknesses, vulnerabilities, and attack patterns makes the understanding of contractors' assurance actions easier and more consistent and offers opportunities for reuse of that approach to provide similar confidence in other systems needed and in other contractors. Confidence is built in the software system through static analysis, dynamic analysis, design inspection, attack surface analysis, threat analysis and modeling, code inspections, and penetration testing during the entire development process. Each of these activities identifies weaknesses and vulnerabilities in the software and its design and allows early correction at minimal cost and time.

Determining whether a system "functions as intended" requires both showing through testing that the intended functionality is there and through test coverage and metrics understanding the system does not perform unrequired functions. As with the confidence measures discussed above, there are several points where insights into the risks and the mitigation of those risks regarding a system's ability to "function as intended" can be obtained. We assert that the software functions as intended, and only as intended, through application of attack patterns/threat emulation, penetration testing, test coverage, and through the use of multiple-redundant implementations of critical elements by different suppliers and having the system failover to an alternate implementation. Each of these activities allows us to identify that the software is indeed functioning as intended without functioning, or being made to function, in unintended ways.

As previously stated, it is common practice in industry to refer to the Common Weakness Enumeration (CWE) catalogue when assessing whether software is "free of vulnerabilities." This approach allows others to understand both what was "looked for" and what was not but could have been "looked for." Similarly, for commercial software packages (proprietary and open source) used as part of a system, the collection of publicly known vulnerabilities in these types of software, called the Common Vulnerabilities and Exposures (CVE) dictionary or the National Vulnerability Database (NVD), are almost always used as reference sources to determine if known issues have been mitigated. When trying to assert that the software is free of vulnerabilities by validating that those CVE and CWE items that are most dangerous to the mission are absent from the software and that the software operates at the least privilege required to complete its task. Therefore, contractors should be required to assure relevant and dangerous CVEs/CWEs are not present in the software.

When software is being developed by a contractor, the system owner may be in a position of verifying the software security. In these cases, it is important to establish requirements, deliverables, and review milestones in ways that ensure high standards for secure coding. There are several requirements and deliverables that will help ensure secure coding practices. There should be a requirement that the contractor develops and delivers a secure coding guidelines document. That document should implement industry best practices for secure coding. Ideally, there should be a process for the government to review and request revisions to that document.

The contractor should provide a full delivery of all source code. The source code delivery should have everything needed to build the code including COTS and FOSS libraries, build scripts, and build instructions. This helps ensure maintainability but also enables independent static and dynamic testing.

The contractor should perform static code analysis on a regular basis. Sending results of static analysis to developers quickly ensures the intent and use of the code is fresh in their minds, and it will be easy to see if they accidentally introduced a weakness that must be fixed. If the contractor uses a DevSecOps process, static and dynamic analysis tools should be part of the build pipeline. The results from the static analyzer should be reviewed regularly and shared with the government. The government should have a place on the contractor's review board that adjudicates the security findings. The government should have a means to report its own security findings.

The government should take an active role in scanning and testing the software deliveries. At major milestone deliveries, the source code should be independently scanned using the government's choice of tools to ensure coverage of security categories described in section 5.2. The software should be dynamically tested, and penetration tested in a lab environment and replicates the operational environment as closely as possible. The end result of all this effort typically results in acceptance and certification from the system owner/government organization.

8 Software Security Authorization and/or Certification

Government organizations and commercial systems alike often leverage authorization and/or certification processes to gain confidence in a system and that the system is free of vulnerabilities. Software certifications are only as good as the criteria in which it is being certified against. Gaining an Authorization to Operate (ATO) in the context of a government information systems does not assure the system or software is free of vulnerabilities. The intention of these processes and certifications are meant to ensure that is the case or if the vulnerabilities present do not introduce too much risk. However, often times the outcome from the certification process does not garner the necessary technical analysis needed to make such claims. With software certification programs, it comes down to the standard in which it is being measured and what analytical methods are used to gain assurance to the standard. The point of this section is to merely articulate to not blindly accept software certification as the panacea. Request ample details on the criteria/rubric in which was used to certify the software as well as the evidence collected to validate the standard was met. Often times these certification processes are paperwork exercises that are not grounded in technical truth. Be cautious and leverage certification where possible, but do not blindly accept that if a software product was "approved for use" by one agency that it meets your standard as your risk profile may be different. The risk one mission owner is willing to accept is often different than what another mission owner will accept.

9 Maintenance and Sustainment

Technology is constantly changing, and to ensure security, software systems must be dynamic and adaptive to current technology and security practices. Software sustainment and maintenance are often used interchangeably, but the two entail different measures. IEEE Standard Glossary of Software Engineering Terminology defines software maintenance as "the process of modifying a

software system or component after deliver to correct faults, improve performance or other attributes, or adapt to a change environment." Software sustainment represents "the processes, procedures, people, material, and information required to support, maintain, and operate the software aspects of a system." (Lapham & Carol, 2006) The key difference is: maintenance is actively modifying the product, whereas sustainment represents the process and measures for securing software.

9.1 Vulnerability Assessments

Regular vulnerability assessments and penetration tests are useful for discovering weaknesses in systems that may arise from misconfigurations, newly discovered vulnerabilities that have not been patched, and changes from configuration-controlled baseline. Vulnerability and penetration testing are used widely in commercial environments, and it is highly recommended for their use more widely within space systems due to their effectiveness.

A penetration test is a type of security test meant to mimic a real-world attack. In a penetration test, cybersecurity professionals will breach computer network, execute exploits, and otherwise attempt to access as much information as possible. At the end of the test, a report is provided discussing the issues penetration testers have found as well as steps that must be undertaken to secure the system.

A vulnerability assessment is performed with the cooperation of the system owners in a more cooperative fashion than a penetration test. In practice, vulnerability assessments are more effective at finding and demonstrating weaknesses than penetration tests. Due to the short timeframe typically allowed, cooperation with system owners to understand the system's operation and obtain credentials leads to more productive testing. In this section, vulnerability assessment and penetration test will be used interchangeably.

9.1.1 Vulnerability Assessment Requirements

Before a test can begin, the scope of the test must be defined. For instance, certain systems may have sensitive data that the penetration testers should not view. If that is the case, the penetration tester shouldn't be barred from accessing the system; instead, testers should be notified of what data can or cannot be accessed or dummy data secured in an identical fashion as the real data should be implanted for the penetration testers to attempt to access.

The penetration tester should be provided the topology—a graphical overview of a computer network—to have the best results. A common misconception is that a genuine attacker would not have network topology at hand when performing their breach; however, an adversary will spend many months probing a network and developing a full topology before beginning their attack. On the other hand, a penetration tester will only have two weeks at most, hence providing topology is a requirement for the most realistic test. Additionally, even if attackers are unable to probe the system, espionage or data breaches can cause the original network topology to be leaked to attackers regardless of their ability to scan the network.

9.2 COTS and FOSS

When choosing to integrate FOSS or COTS into software development, project managers must understand that they take responsibility for security weaknesses that may exist or arise in the future in those products. As mentioned in Section 3.3, corporate or government-based procedures may

not have been followed to the same standard as the company using the software. This could entail the company acquiring the software to provide additional measures to ensure that the FOSS or COTS product adheres to company production standards and qualifications. Though adopting a COTS or FOSS product into the software development cycle may reduce in-house development costs, extraneous measures are required to ensure the integrity and security of the external software product.

Programs should include patch management in their maintenance and sustainment plan. The patch management plan should address all COTS and FOSS packages included in the system including complete software packages and libraries. Documenting and maintaining configuration control of the list of packages used within the system is essential to ensuring the patch management plan covers all the included software. For complex systems, there is software that can assist with patch management such as HEAT and Tanium (Ivanti software, n.d.), (Tanium Software, n.d.).

As to determining the preference between using COTS versus FOSS in terms of sustainment and maintenance, there is no clear answer. Each option must be weighed depending on the needs specific software system as well as the needs of the company. COTS can lower development costs but increase maintenance costs. As opposed to in-house developed tools, COTS would require extra care in dealing with vendor relations of the COTS product. Its maintenance cost is determined security is composed of patching, robust testing, and its integration to company processes, protocols, and standards. Its sustainment is more complicated due to increased use of COTS software products. While the process for sustaining legacy software products is well defined, sustaining modern commercial software poses more complications. There are two main types of COTS: a COTS-solution system and COTS-aggregate system. The first describes a product or suite of products from a single vendor that can be altered to a customer's needs, while the later entails multiple products developed from multiple vendors. COTS-solution software requires the developers to rely on one relationship with a vendor, whereas COTS-aggregate solutions require sustained relationships between multiple vendors. Relationships should be created with companies that have a great understanding of COTS maintained and sustainment, otherwise high risk is imposed on the company inheriting the COTS software. When dealing with sustainment, a company needs to consider how to manage potential system obsolesce, technology refresh, source code escrow, licensing management, and COTS-aggregate system architectures.

In terms of maintenance, COTS cost includes patching, robust testing, adopting company development processes, protocols, and standards. Depending on the COTS, it may also have FOSS components that must be properly maintained. A maintenance plan should be discussed with the vendor to determine security measures done at their end. This plan should also factor in an intended lifeline of the software product inheriting the COTS, to ensure all external components will be sustained through the duration of the life of the product.

FOSS is a unique and has its own measures for sustainment and maintenance. As FOSS is often done voluntarily by the efforts of many developers, security measures might fall mostly or completely on the organization. This would include, when source code is available, complete sustainment and maintenance done at the company level. Using software to manage software libraries and packages is particular useful in determining if is software is securely up-to-date, as most software patches fix security bugs (refer to section 3.3). While this does provide more control over the software product from a maintenance perspective, it also requires more work upon the

company. Some FOSS do have their own maintenance and sustainment procedures, and regularly provide patch updates. Each product needs to be evaluated individually in terms of maintenance and sustainment, as no FOSS product is treated equal to another.

APPENDIX A: HIGH PRIORITY COMMON WEAKNESS ENUMERATIONS

Research was performed at NASA's IV&V Program, based on software analysis of many projects from various NASA centers. Throughout this research, the analysts have found certain types of flaws recurring in several different missions in high volume. To understand the flaws, it is important to first understand how they are labeled and categorized. MITRE has implemented a system to assign an ID (CWE_ID) to each weakness. This system of assigning IDs to weaknesses helps scanning tools easily identify and report whether a given weakness is present in the developer's system. According to their site, CWE is:

> *"…a unified, measurable set of software weaknesses that enables the effective discussion, description, selection, and use of software security tools and services that can find these weaknesses in source code and operational systems. The CWE also enables better understanding and management of software weaknesses related to architecture and design. It enumerates design and architectural weaknesses, as well as low-level coding and design errors."*

The CWE lists hundreds of weaknesses. For each weakness, the site provides a definition, consequence of the weakness (one or more), detection methods, potential remediation advice, and a common weakness scoring system (CWSS) value. The CWSS is an ongoing effort by MITRE to provide a mechanism for prioritizing software weaknesses in a consistent manner. A modified version of CWSS was used to prioritize CWEs and are listed below. Some developers currently rely on the weakness ranking subjectively outputted by MITRE or the various scanning tools used to check code. The problem is these tools do not have the necessary knowledge of space systems to determine which weaknesses are more or less important to mission. A prioritization and categorization have been performed below. The domain applicability means if the CWE is a high priority CWE for either the ground system, spacecraft, or both. The list below is in order of priority from highest to lowest, but all the listed 386 CWEs are considered high priority (i.e., priority one CWEs) for space systems but they are listed in descending priority order. The CWE types (variant, base, class, etc.) are defined in https://cwe.mitre.org/documents/glossary/

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-020	Improper Input Validation	Class	Spacecraft & Ground
CWE-041	Improper Resolution of Path Equivalence	Base	Ground
CWE-074	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	Class	Ground
CWE-089	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Base	Ground
CWE-098	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')	Base	Ground
CWE-285	Improper Authorization	Class	Spacecraft & Ground
CWE-078	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	Base	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-079	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	Base	Ground
CWE-080	Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	Variant	Ground
CWE-059	Improper Link Resolution Before File Access ('Link Following')	Base	Ground
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	Base	Spacecraft & Ground
CWE-094	Improper Control of Generation of Code ('Code Injection')	Class	Ground
CWE-302	Authentication Bypass by Assumed-Immutable Data	Variant	Ground
CWE-088	Argument Injection or Modification	Base	Ground
CWE-697	Incorrect Comparison	Class	Spacecraft & Ground
CWE-095	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')	Base	Ground
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Class	Spacecraft & Ground
CWE-073	External Control of File Name or Path	Class	Ground
CWE-642	External Control of Critical State Data	Class	Ground
CWE-680	Integer Overflow to Buffer Overflow	Chain	Spacecraft & Ground
CWE-311	Missing Encryption of Sensitive Data	Base	Ground
CWE-096	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')	Base	Ground
CWE-427	Uncontrolled Search Path Element	Base	Ground
CWE-267	Privilege Defined With Unsafe Actions	Base	Ground
CWE-312	Cleartext Storage of Sensitive Information	Variant	Ground
CWE-350	Reliance on Reverse DNS Resolution for a Security-Critical Action	Variant	Ground
CWE-276	Incorrect Default Permissions	Variant	Ground
CWE-180	Incorrect Behavior Order: Validate Before Canonicalize	Base	Ground
CWE-093	Improper Neutralization of CRLF Sequences ('CRLF Injection')	Base	Ground
CWE-473	PHP External Variable Modification	Variant	Ground
CWE-269	Improper Privilege Management	Class	Ground
CWE-023	Relative Path Traversal	Base	Spacecraft & Ground
CWE-190	Integer Overflow or Wraparound	Base	Spacecraft & Ground
CWE-022	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	Class	Spacecraft & Ground
CWE-829	Inclusion of Functionality from Untrusted Control Sphere	Class	Ground
CWE-272	Least Privilege Violation	Base	Ground
CWE-426	Untrusted Search Path	Base	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-118	Incorrect Access of Indexable Resource ('Range Error')	Class	Spacecraft & Ground
CWE-130	Improper Handling of Length Parameter Inconsistency	Base	Spacecraft & Ground
CWE-122	Heap-based Buffer Overflow	Variant	Spacecraft & Ground
CWE-090	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')	Base	Ground
CWE-425	Direct Request ('Forced Browsing')	Base	Ground
CWE-129	Improper Validation of Array Index	Base	Spacecraft & Ground
CWE-113	Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')	Base	Ground
CWE-862	Missing Authorization	Class	Ground
CWE-250	Execution with Unnecessary Privileges	Class	Ground
CWE-400	Uncontrolled Resource Consumption	Class	Ground
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	Class	Spacecraft & Ground
CWE-116	Improper Encoding or Escaping of Output	Class	Ground
CWE-131	Incorrect Calculation of Buffer Size	Base	Spacecraft & Ground
CWE-270	Privilege Context Switching Error	Base	Ground
CWE-181	Incorrect Behavior Order: Validate Before Filter	Base	Ground
CWE-472	External Control of Assumed-Immutable Web Parameter	Base	Ground
CWE-294	Authentication Bypass by Capture-replay	Base	Spacecraft & Ground
CWE-184	Incomplete Blacklist	Base	Spacecraft & Ground
CWE-444	Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')	Base	Ground
CWE-602	Client-Side Enforcement of Server-Side Security	Base	Ground
CWE-087	Improper Neutralization of Alternate XSS Syntax	Variant	Ground
CWE-117	Improper Output Neutralization for Logs	Base	Ground
CWE-084	Improper Neutralization of Encoded URI Schemes in a Web Page	Variant	Ground
CWE-805	Buffer Access with Incorrect Length Value	Base	Spacecraft & Ground
CWE-303	Incorrect Implementation of Authentication Algorithm	Base	Spacecraft & Ground
CWE-352	Cross-Site Request Forgery (CSRF)	Composite	Ground
CWE-384	Session Fixation	Composite	Ground
CWE-692	Incomplete Blacklist to Cross-Site Scripting	Chain	Ground
CWE-288	Authentication Bypass Using an Alternate Path or Channel	Base	Spacecraft & Ground
CWE-279	Incorrect Execution-Assigned Permissions	Variant	Ground
CWE-306	Missing Authentication for Critical Function	Variant	Spacecraft & Ground
CWE-732	Incorrect Permission Assignment for Critical Resource	Class	Spacecraft & Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-177	Improper Handling of URL Encoding (Hex Encoding)	Variant	Ground
CWE-138	Improper Neutralization of Special Elements	Class	Spacecraft & Ground
CWE-295	Improper Certificate Validation	Base	Ground
CWE-733	Compiler Optimization Removal or Modification of Security-critical Code	Base	Spacecraft & Ground
CWE-415	Double Free	Variant	Ground
CWE-833	Deadlock	Base	Spacecraft & Ground
CWE-193	Off-by-one Error	Base	Ground
CWE-330	Use of Insufficiently Random Values	Class	Spacecraft & Ground
CWE-441	Unintended Proxy or Intermediary ('Confused Deputy')	Class	Ground
CWE-348	Use of Less Trusted Source	Base	Spacecraft & Ground
CWE-368	Context Switching Race Condition	Base	Spacecraft & Ground
CWE-565	Reliance on Cookies without Validation and Integrity Checking	Base	Ground
CWE-667	Improper Locking	Base	Spacecraft & Ground
CWE-614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	Variant	Ground
CWE-086	Improper Neutralization of Invalid Characters in Identifiers in Web Pages	Variant	Ground
CWE-081	Improper Neutralization of Script in an Error Message Web Page	Variant	Ground
CWE-282	Improper Ownership Management	Class	Ground
CWE-770	Allocation of Resources Without Limits or Throttling	Base	Spacecraft & Ground
CWE-185	Incorrect Regular Expression	Class	Spacecraft & Ground
CWE-436	Interpretation Conflict	Base	Ground
CWE-663	Use of a Non-reentrant Function in a Concurrent Context	Base	Spacecraft & Ground
CWE-287	Improper Authentication	Class	Spacecraft & Ground
CWE-036	Absolute Path Traversal	Base	Ground
CWE-863	Incorrect Authorization	Class	Ground
CWE-297	Improper Validation of Certificate with Host Mismatch	Variant	Ground
CWE-665	Improper Initialization	Class	Spacecraft & Ground
CWE-367	Time-of-check Time-of-use (TOCTOU) Race Condition	Base	Spacecraft & Ground
CWE-638	Not Using Complete Mediation	Class	Ground
CWE-097	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page	Variant	Ground
CWE-674	Uncontrolled Recursion	Base	Spacecraft & Ground
CWE-290	Authentication Bypass by Spoofing	Base	Spacecraft & Ground
CWE-196	Unsigned to Signed Conversion Error	Variant	Spacecraft & Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	Base	Ground
CWE-154	Improper Neutralization of Variable Name Delimiters	Variant	Ground
CWE-209	Information Exposure Through an Error Message	Base	Ground
CWE-158	Improper Neutralization of Null Byte or NUL Character	Variant	Spacecraft & Ground
CWE-693	Protection Mechanism Failure	Class	Spacecraft & Ground
CWE-128	Wrap-around Error	Base	Spacecraft & Ground
CWE-301	Reflection Attack in an Authentication Protocol	Variant	Spacecraft & Ground
CWE-428	Unquoted Search Path or Element	Base	Spacecraft & Ground
CWE-798	Use of Hard-coded Credentials	Base	Spacecraft & Ground
CWE-522	Insufficiently Protected Credentials	Base	Spacecraft & Ground
CWE-346	Origin Validation Error	Base	Spacecraft & Ground
CWE-300	Channel Accessible by Non-Endpoint ('Man-in-the-Middle')	Class	Spacecraft & Ground
CWE-689	Permission Race Condition During Resource Copy	Composite	Ground
CWE-412	Unrestricted Externally Accessible Lock	Base	Ground
CWE-331	Insufficient Entropy	Base	Spacecraft & Ground
CWE-676	Use of Potentially Dangerous Function	Base	Ground
CWE-807	Reliance on Untrusted Inputs in a Security Decision	Base	Ground
CWE-200	Information Exposure	Class	Ground
CWE-314	Cleartext Storage in the Registry	Variant	Ground
CWE-039	Path Traversal: 'C:dirname'	Variant	Ground
CWE-434	Unrestricted Upload of File with Dangerous Type	Base	Spacecraft & Ground
CWE-091	XML Injection (aka Blind XPath Injection)	Base	Ground
CWE-476	NULL Pointer Dereference	Base	Spacecraft & Ground
CWE-459	Incomplete Cleanup	Base	Ground
CWE-772	Missing Release of Resource after Effective Lifetime	Base	Ground
CWE-319	Cleartext Transmission of Sensitive Information	Variant	Spacecraft & Ground
CWE-284	Improper Access Control	Class	Spacecraft & Ground
CWE-315	Cleartext Storage of Sensitive Information in a Cookie	Variant	Ground
CWE-523	Unprotected Transport of Credentials	Variant	Spacecraft & Ground
CWE-146	Improper Neutralization of Expression/Command Delimiters	Variant	Spacecraft & Ground
CWE-083	Improper Neutralization of Script in Attributes in a Web Page	Variant	Ground
CWE-502	Deserialization of Untrusted Data	Variant	Ground
CWE-050	Path Equivalence: '//multiple/leading/slash'	Variant	Ground
CWE-150	Improper Neutralization of Escape, Meta, or Control Sequences	Variant	Spacecraft & Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-662	Improper Synchronization	Base	Spacecraft & Ground
CWE-271	Privilege Dropping / Lowering Errors	Class	Ground
CWE-611	Improper Restriction of XML External Entity Reference ('XXE')	Variant	Ground
CWE-179	Incorrect Behavior Order: Early Validation	Base	Spacecraft & Ground
CWE-028	Path Traversal: '..\filedir'	Variant	Ground
CWE-691	Insufficient Control Flow Management	Class	Spacecraft & Ground
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	Base	Spacecraft & Ground
CWE-134	Use of Externally-Controlled Format String	Base	Spacecraft & Ground
CWE-099	Improper Control of Resource Identifiers ('Resource Injection')	Base	Ground
CWE-037	Path Traversal: '/absolute/pathname/here'	Variant	Ground
CWE-241	Improper Handling of Unexpected Data Type	Base	Spacecraft & Ground
CWE-416	Use After Free	Base	Spacecraft & Ground
CWE-257	Storing Passwords in a Recoverable Format	Base	Ground
CWE-521	Weak Password Requirements	Base	Ground
CWE-318	Cleartext Storage of Sensitive Information in Executable	Variant	Ground
CWE-046	Path Equivalence: 'filename ' (Trailing Space)	Variant	Ground
CWE-334	Small Space of Random Values	Base	Ground
CWE-261	Weak Cryptography for Passwords	Variant	Ground
CWE-038	Path Traversal: '\absolute\pathname\here'	Variant	Ground
CWE-341	Predictable from Observable State	Base	Ground
CWE-082	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page	Variant	Ground
CWE-695	Use of Low-Level Functionality	Base	Spacecraft & Ground
CWE-648	Incorrect Use of Privileged APIs	Base	Ground
CWE-125	Out-of-bounds Read	Base	Ground
CWE-262	Not Using Password Aging	Variant	Ground
CWE-263	Password Aging with Long Expiration	Base	Ground
CWE-187	Partial String Comparison	Variant	Ground
CWE-062	UNIX Hard Link	Variant	Ground
CWE-085	Doubled Character XSS Manipulations	Variant	Ground
CWE-032	Path Traversal: '...' (Triple Dot)	Variant	Ground
CWE-157	Failure to Sanitize Paired Delimiters	Variant	Spacecraft & Ground
CWE-172	Encoding Error	Class	Spacecraft & Ground
CWE-943	Improper Neutralization of Special Elements in Data Query Logic	Class	Ground
CWE-824	Access of Uninitialized Pointer	Base	Ground
CWE-173	Improper Handling of Alternate Encoding	Variant	Spacecraft & Ground
CWE-842	Placement of User into Incorrect Group	Base	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	Base	Ground
CWE-564	SQL Injection: Hibernate	Variant	Ground
CWE-307	Improper Restriction of Excessive Authentication Attempts	Base	Spacecraft & Ground
CWE-681	Incorrect Conversion between Numeric Types	Class	Ground
CWE-234	Failure to Handle Missing Parameter	Variant	Ground
CWE-363	Race Condition Enabling Link Following	Base	Ground
CWE-075	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)	Class	Ground
CWE-064	Windows Shortcut Following (.LNK)	Variant	Ground
CWE-266	Incorrect Privilege Assignment	Base	Ground
CWE-488	Exposure of Data Element to Wrong Session	Variant	Ground
CWE-077	Improper Neutralization of Special Elements used in a Command ('Command Injection')	Class	Spacecraft & Ground
CWE-061	UNIX Symbolic Link (Symlink) Following	Composite	Spacecraft & Ground
CWE-664	Improper Control of a Resource Through its Lifetime	Class	Spacecraft & Ground
CWE-640	Weak Password Recovery Mechanism for Forgotten Password	Base	Ground
CWE-042	Path Equivalence: 'filename.' (Trailing Dot)	Variant	Ground
CWE-057	Path Equivalence: 'fakedir/../realdir/filename'	Variant	Ground
CWE-370	Missing Check for Certificate Revocation after Initial Check	Variant	Ground
CWE-494	Download of Code Without Integrity Check	Base	Spacecraft & Ground
CWE-539	Information Exposure Through Persistent Cookies	Variant	Ground
CWE-603	Use of Client-Side Authentication	Base	Ground
CWE-757	Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')	Class	Ground
CWE-277	Insecure Inherited Permissions	Variant	Ground
CWE-918	Server-Side Request Forgery (SSRF)	Base	Ground
CWE-040	Path Traversal: '\\UNC\share\name\' (Windows UNC Share)	Variant	Ground
CWE-248	Uncaught Exception	Base	Spacecraft
CWE-203	Information Exposure Through Discrepancy	Class	Ground
CWE-033	Path Traversal: '....' (Multiple Dot)	Variant	Ground
CWE-049	Path Equivalence: 'filename/' (Trailing Slash)	Variant	Ground
CWE-055	Path Equivalence: '/./' (Single Dot Directory)	Variant	Ground
CWE-291	Reliance on IP Address for Authentication	Variant	Ground
CWE-015	External Control of System or Configuration Setting	Base	Spacecraft & Ground
CWE-349	Acceptance of Extraneous Untrusted Data With Trusted Data	Base	Spacecraft & Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-006	J2EE Misconfiguration: Insufficient Session-ID Length	Variant	Ground
CWE-749	Exposed Dangerous Method or Function	Base	Ground
CWE-147	Improper Neutralization of Input Terminators	Variant	Ground
CWE-372	Incomplete Internal State Distinction	Base	Spacecraft & Ground
CWE-321	Use of Hard-coded Cryptographic Key	Base	Ground
CWE-299	Improper Check for Certificate Revocation	Base	Ground
CWE-191	Integer Underflow (Wrap or Wraparound)	Base	Spacecraft & Ground
CWE-656	Reliance on Security Through Obscurity	Base	Ground
CWE-825	Expired Pointer Dereference	Base	Ground
CWE-788	Access of Memory Location After End of Buffer	Base	Ground
CWE-170	Improper Null Termination	Base	Spacecraft & Ground
CWE-347	Improper Verification of Cryptographic Signature	Base	Ground
CWE-916	Use of Password Hash With Insufficient Computational Effort	Base	Ground
CWE-144	Improper Neutralization of Line Delimiters	Variant	Ground
CWE-908	Use of Uninitialized Resource	Base	Ground
CWE-538	File and Directory Information Exposure	Base	Ground
CWE-620	Unverified Password Change	Variant	Ground
CWE-404	Improper Resource Shutdown or Release	Class	Ground
CWE-707	Improper Enforcement of Message or Data Structure	Class	Spacecraft & Ground
CWE-328	Reversible One-Way Hash	Base	Ground
CWE-326	Inadequate Encryption Strength	Class	Spacecraft & Ground
CWE-706	Use of Incorrectly-Resolved Name or Reference	Class	Ground
CWE-353	Missing Support for Integrity Check	Base	Spacecraft & Ground
CWE-325	Missing Required Cryptographic Step	Base	Spacecraft & Ground
CWE-013	ASP.NET Misconfiguration: Password in Configuration File	Variant	Ground
CWE-244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	Variant	Ground
CWE-823	Use of Out-of-range Pointer Offset	Base	Spacecraft & Ground
CWE-456	Missing Initialization of a Variable	Base	Spacecraft & Ground
CWE-268	Privilege Chaining	Base	Ground
CWE-259	Use of Hard-coded Password	Base	Ground
CWE-197	Numeric Truncation Error	Base	Ground
CWE-289	Authentication Bypass by Alternate Name	Variant	Ground
CWE-424	Improper Protection of Alternate Path	Class	Spacecraft & Ground
CWE-649	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking	Base	Ground
CWE-822	Untrusted Pointer Dereference	Base	Spacecraft & Ground
CWE-942	Overly Permissive Cross-domain Whitelist	Variant	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-140	Improper Neutralization of Delimiters	Base	Spacecraft & Ground
CWE-124	Buffer Underwrite ('Buffer Underflow')	Base	Spacecraft & Ground
CWE-065	Windows Hard Link	Variant	Ground
CWE-029	Path Traversal: '\..\filename'	Variant	Ground
CWE-035	Path Traversal: '.../...//'	Variant	Ground
CWE-056	Path Equivalence: 'filedir*' (Wildcard)	Variant	Ground
CWE-256	Unprotected Storage of Credentials	Variant	Ground
CWE-838	Inappropriate Encoding for Output Context	Base	Ground
CWE-366	Race Condition within a Thread	Base	Spacecraft & Ground
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')	Variant	Ground
CWE-213	Intentional Information Exposure	Base	Ground
CWE-233	Improper Handling of Parameters	Base	Spacecraft & Ground
CWE-058	Path Equivalence: Windows 8.3 Filename	Variant	Ground
CWE-407	Algorithmic Complexity	Base	Spacecraft & Ground
CWE-305	Authentication Bypass by Primary Weakness	Base	Ground
CWE-204	Response Discrepancy Information Exposure	Base	Ground
CWE-510	Trapdoor	Base	Ground
CWE-514	Covert Channel	Class	Ground
CWE-915	Improperly Controlled Modification of Dynamically-Determined Object Attributes	Base	Ground
CWE-027	Path Traversal: 'dir/../../filename'	Variant	Ground
CWE-031	Path Traversal: 'dir\..\..\filename'	Variant	Ground
CWE-034	Path Traversal: '....//'	Variant	Ground
CWE-043	Path Equivalence: 'filename....' (Multiple Trailing Dot)	Variant	Ground
CWE-051	Path Equivalence: '/multiple//internal/slash'	Variant	Ground
CWE-052	Path Equivalence: '/multiple/trailing/slash//'	Variant	Ground
CWE-054	Path Equivalence: 'filedir\' (Trailing Backslash)	Variant	Ground
CWE-296	Improper Following of a Certificate's Chain of Trust	Base	Ground
CWE-176	Improper Handling of Unicode Encoding	Variant	Spacecraft & Ground
CWE-114	Process Control	Base	Ground
CWE-941	Incorrectly Specified Destination in a Communication Channel	Base	Ground
CWE-799	Improper Control of Interaction Frequency	Class	Ground
CWE-940	Improper Verification of Source of a Communication Channel	Base	Ground
CWE-786	Access of Memory Location Before Start of Buffer	Base	Spacecraft & Ground
CWE-178	Improper Handling of Case Sensitivity	Base	Ground
CWE-149	Improper Neutralization of Quoting Syntax	Variant	Ground
CWE-403	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')	Base	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-911	Improper Update of Reference Count	Base	Ground
CWE-841	Improper Enforcement of Behavioral Workflow	Base	Ground
CWE-784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision	Variant	Ground
CWE-754	Improper Check for Unusual or Exceptional Conditions	Class	Ground
CWE-358	Improperly Implemented Security Check for Standard	Base	Ground
CWE-219	Sensitive Data Under Web Root	Variant	Ground
CWE-226	Sensitive Information Uncleared Before Release	Base	Spacecraft & Ground
CWE-235	Improper Handling of Extra Parameters	Variant	Ground
CWE-048	Path Equivalence: 'file name' (Internal Whitespace)	Variant	Ground
CWE-525	Information Exposure Through Browser Caching	Variant	Ground
CWE-593	Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created	Variant	Ground
CWE-304	Missing Critical Step in Authentication	Base	Ground
CWE-005	J2EE Misconfiguration: Data Transmission Without Encryption	Variant	Ground
CWE-182	Collapse of Data into Unsafe Value	Base	Ground
CWE-313	Cleartext Storage in a File or on Disk	Variant	Ground
CWE-183	Permissive Whitelist	Base	Spacecraft & Ground
CWE-258	Empty Password in Configuration File	Variant	Ground
CWE-914	Improper Control of Dynamically-Identified Variables	Base	Ground
CWE-151	Improper Neutralization of Comment Delimiters	Variant	Ground
CWE-409	Improper Handling of Highly Compressed Data (Data Amplification)	Base	Spacecraft & Ground
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	Base	Spacecraft & Ground
CWE-1004	Sensitive Cookie Without 'HttpOnly' Flag	Variant	Ground
CWE-273	Improper Check for Dropped Privileges	Base	Ground
CWE-401	Improper Release of Memory Before Removing Last Reference	Variant	Ground
CWE-610	Externally Controlled Reference to a Resource in Another Sphere	Class	Spacecraft & Ground
CWE-155	Improper Neutralization of Wildcards or Matching Symbols	Variant	Ground
CWE-698	Execution After Redirect (EAR)	Base	Ground
CWE-410	Insufficient Resource Pool	Base	Spacecraft & Ground
CWE-167	Improper Handling of Additional Special Element	Base	Ground
CWE-156	Improper Neutralization of Whitespace	Variant	Ground
CWE-457	Use of Uninitialized Variable	Variant	Spacecraft & Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-939	Improper Authorization in Handler for Custom URL Scheme	Base	Ground
CWE-696	Incorrect Behavior Order	Class	Spacecraft & Ground
CWE-014	Compiler Removal of Code to Clear Buffers	Variant	Ground
CWE-111	Direct Use of Unsafe JNI	Base	Ground
CWE-239	Failure to Handle Incomplete Element	Variant	Ground
CWE-382	J2EE Bad Practices: Use of System.exit()	Variant	Ground
CWE-493	Critical Public Variable Without Final Modifier	Variant	Ground
CWE-497	Exposure of System Data to an Unauthorized Control Sphere	Variant	Ground
CWE-166	Improper Handling of Missing Special Element	Base	Ground
CWE-471	Modification of Assumed-Immutable Data (MAID)	Base	Ground
CWE-455	Non-exit on Failed Initialization	Base	Ground
CWE-364	Signal Handler Race Condition	Base	Spacecraft & Ground
CWE-142	Improper Neutralization of Value Delimiters	Variant	Ground
CWE-236	Improper Handling of Undefined Parameters	Variant	Ground
CWE-359	Exposure of Private Information ('Privacy Violation')	Class	Ground
CWE-230	Improper Handling of Missing Values	Variant	Ground
CWE-214	Information Exposure Through Process Environment	Variant	Ground
CWE-479	Signal Handler Use of a Non-reentrant Function	Variant	Ground
CWE-123	Write-what-where Condition	Base	Spacecraft & Ground
CWE-336	Same Seed in Pseudo-Random Number Generator (PRNG)	Base	Ground
CWE-069	Improper Handling of Windows ::DATA Alternate Data Stream	Variant	Ground
CWE-451	User Interface (UI) Misrepresentation of Critical Information	Class	Ground
CWE-322	Key Exchange without Entity Authentication	Base	Ground
CWE-323	Reusing a Nonce, Key Pair in Encryption	Base	Ground
CWE-554	ASP.NET Misconfiguration: Not Using Input Validation Framework	Variant	Ground
CWE-345	Insufficient Verification of Data Authenticity	Class	Ground
CWE-242	Use of Inherently Dangerous Function	Base	Ground
CWE-708	Incorrect Ownership Assignment	Base	Ground
CWE-755	Improper Handling of Exceptional Conditions	Class	Ground
CWE-126	Buffer Over-read	Variant	Ground
CWE-397	Declaration of Throws for Generic Exception	Base	Spacecraft & Ground
CWE-617	Reachable Assertion	Variant	Ground
CWE-067	Improper Handling of Windows Device Names	Variant	Ground
CWE-421	Race Condition During Access to Alternate Channel	Base	Ground
CWE-030	Path Traversal: '\dir\..\filename'	Variant	Ground

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-828	Signal Handler with Functionality that is not Asynchronous-Safe	Base	Ground
CWE-621	Variable Extraction Error	Base	Ground
CWE-690	Unchecked Return Value to NULL Pointer Dereference	Chain	Spacecraft & Ground
CWE-274	Improper Handling of Insufficient Privileges	Base	Ground
CWE-783	Operator Precedence Logic Error	Variant	Ground
CWE-283	Unverified Ownership	Base	Ground
CWE-201	Information Exposure Through Sent Data	Variant	Ground
CWE-567	Unsynchronized Access to Shared Data in a Multithreaded Context	Base	Spacecraft & Ground
CWE-705	Incorrect Control Flow Scoping	Class	Ground
CWE-141	Improper Neutralization of Parameter/Argument Delimiters	Variant	Ground
CWE-153	Improper Neutralization of Substitution Characters	Variant	Ground
CWE-232	Improper Handling of Undefined Values	Variant	Ground
CWE-317	Cleartext Storage of Sensitive Information in GUI	Variant	Ground
CWE-639	Authorization Bypass Through User-Controlled Key	Base	Ground
CWE-1021	Improper Restriction of Rendered UI Layers or Frames	Base	Ground
CWE-618	Exposed Unsafe ActiveX Method	Base	Ground
CWE-152	Improper Neutralization of Macro Symbols	Variant	Ground
CWE-281	Improper Preservation of Permissions	Base	Ground
CWE-369	Divide By Zero	Base	Spacecraft & Ground
CWE-252	Unchecked Return Value	Base	Ground
CWE-208	Information Exposure Through Timing Discrepancy	Base	Ground
CWE-215	Information Exposure Through Debug Information	Variant	Ground
CWE-356	Product UI does not Warn User of Unsafe Actions	Base	Ground
CWE-589	Call to Non-ubiquitous API	Variant	Spacecraft
CWE-198	Use of Incorrect Byte Ordering	Base	Spacecraft
CWE-396	Declaration of Catch for Generic Exception	Base	Spacecraft
CWE-489	Leftover Debug Code	Base	Spacecraft & Ground
CWE-112	Missing XML Validation	Base	Spacecraft & Ground
CWE-764	Multiple Locks of a Critical Resource	Variant	Spacecraft & Ground
CWE-789	Uncontrolled Memory Allocation	Variant	Spacecraft & Ground
CWE-405	Asymmetric Resource Consumption (Amplification)	Class	Spacecraft
CWE-413	Improper Resource Locking	Base	Spacecraft
CWE-508	Non-Replicating Malicious Code	Base	Spacecraft
CWE-511	Logic/Time Bomb	Base	Spacecraft
CWE-308	Use of Single-factor Authentication	Base	Spacecraft & Ground
CWE-466	Return of Pointer Value Outside of Expected Range	Base	Spacecraft

CWE ID	CWE Name	CWE Type	Domain Applicability
CWE-787	Out-of-bounds Write	Base	Spacecraft
CWE-506	Embedded Malicious Code	Class	Spacecraft
CWE-763	Release of Invalid Pointer or Reference	Base	Spacecraft
CWE-924	Improper Enforcement of Message Integrity During Transmission in a Communication Channel	Class	Spacecraft
CWE-121	Stack-based Buffer Overflow	Variant	Spacecraft
CWE-192	Integer Coercion Error	Class	Spacecraft
CWE-468	Incorrect Pointer Scaling	Base	Spacecraft
CWE-469	Use of Pointer Subtraction to Determine Size	Base	Spacecraft
CWE-834	Excessive Iteration	Base	Spacecraft

APPENDIX B: SECURE CODING DESIGN REVIEW CHECKLIST

The source of this information is from NASA's Software Assurance Research Program (SARP) initiative on secure software development.

Language

☐ Examine the pros and cons for the selected language for issues such as memory management and exception handling.

Libraries and Frameworks

☐ Ensure that abstraction libraries are being used to simplify the code, mitigate risky APIs, provide separation between the data and code, and provide memory management.

☐ Verify that the design properly separates code and data.

Authentication

☐ Verify that the plan for handling initial account passwords uses a secure mechanism for creation and distribution of account credentials.

☐ Ensure that any stored authentication credentials are outside of the code, encrypted, and in a secure location.

☐ Verify that credentials for back-end connections are limited to minimal actions and use generated passwords valid for specified time intervals.

Encryption

☐ Ensure that all sensitive data and resources are identified and encrypted.

☐ Verify that the transmission of all sensitive data in the system is encrypted.

☐ Verify that the cryptographic algorithms being used are current and strong.

Error Handling

☐ Ensure that the program is designed in such a way that it will always fail gracefully (fail closed).

External Communication

☐ Ensure that all data used in external commands is properly protected.

☐ Verify that all communications use strictly defined communication protocols.

☐ Verify that checksums are used for all data file transmissions.

External Resources

☐ Ensure that the minimum and maximum expectations for resources are specified and behaviors are defined for when those limits are approached or exceeded.

☐ Verify that the interleaving of operations on files from multiple processes is minimized.

☐ Verify that proper locking mechanisms are present on resources.

☐ Ensure that deadlock is actively being prevented for in the design of the code.

☐ Ensure that throttling mechanisms are designed to prevent resource exhaustion.

Input Validation
☐ Verify that all input from external sources is validated.
☐ Ensure that input validation uses whitelists (not blacklists).
☐ Ensure that input validation is duplicated on both the client and the server side (web only).

Segregation
☐ Verify that the design segregates functionality by access level.
☐ Ensure that the design compartmentalizes sensitive data into “safe” areas.
☐ Verify that the design supports running in sandbox environments to ensure safe interaction between the software and the operating system.

Threat Protection
☐ Verify that countermeasures are sufficient for all threats and attacks identified in threat modeling.
☐ Perform attack modeling from an attacker’s perspective to identify weaknesses and vulnerabilities that should be countered. Ensure that the design is not vulnerable to these attacks.

Appendix C: Secure Coding Code Review Checklist

The source of this information is from NASA's Software Assurance Research Program (SARP) initiative on secure software development.

☐ **Buffer Copy without Checking Size of Input**

- Are the sizes of all input buffers less than the sizes of their corresponding output buffers?
- Do all user inputs meet the requirements of the buffer?
  - If the array is only 24 characters, do not allow a character array of 25 characters

☐ **Buffer Access with Incorrect Length Value**

- Is the length value used for reading/writing to a buffer within the bounds of the buffer?

☐ **Cleartext Storage in the Registry**

- If storing sensitive information in the registry, is the information encrypted and not just encoded to be human-unreadable.

☐ **Cleartext Storage of Sensitive Information in a Cookie**

- If storing sensitive information in a cookie, is the information encrypted and not just encoded to be human-unreadable.

☐ **Cleartext Storage of Sensitive Information in Executable**

- If storing sensitive information in an executable, is the information encrypted and not just encoded to be human-unreadable.

☐ **Out-of-bounds Write**

- Are all pointers and index values within the bounds of the intended buffer?
- Are values passed to functions like memcpy() within the bounds of the buffer being assigned to?

☐ **Execution with Unnecessary Privileges**

- Are permission levels/privileges at their minimum whenever possible?
- Does the code executed during times of increased privilege handle exceptions correctly to ensure the subsequent call to downgrade permissions?
- Are all processes and threads called by processes using minimum privileges necessary?
  - If the parent process is running as root, all child threads will have root permissions.

☐ **Use of Externally-Controlled Format String**

- Are format strings avoided when dealing with external input?
- If using external input with particularly dangerous functions (printf() for example), is the formatting is done by the program, and not the user?

  For example:
  Printf(y,userInput) vs printf(userInput). The latter case allows the user to specify format strings such as %n or %s which can lead to a variety of different attacks.

☐ **Allocation of Resources Without Limits or Throttling**

- Are limits set on how many resources can be allocated to a specific user and in total to prevent resource exhaustion?

☐ **Improper Input Validation, especially custom input validation to enforce business rules**

- Are whitelists used instead of blacklists to validate known inputs?
- Does the input validation verify the values of the inputs and not just the characters?
    - For instance, if a color is expected and the user provides “boat”, although it may syntactically be correct (alphabetic characters and within a certain limit), its clearly not the correct input.

☐ **Improper Limitation of a Pathname to a Restricted Directory**

- Do whitelists for the types of characters that can be included in the pathname only allow one “.” or prevent the usage of “/”?
    - For example you may have that a string must start with “/safe/directory” to be at the start of an input, but if the user inputs “safe/directory/../../badStuff”, the system has been corrupted.
- Does the input validation prevent the insertion of null characters?
    - Null characters can prevent normal functionality.

☐ **Improper Neutralization of Special Elements used in an OS Command**

- When input is being processed and used in any system level commands, are all special elements being neutralized?
    - For instance, if user input will be appended to a system level command such as running a program, ensure that it is properly formatted and does not contain special characters like &, /, .. , or other special elements that can lead to malicious functionality.

☐ **Improper Restriction of Operations within the Bounds of a Memory Buffer**

- Is all input within the bounds of the buffers being created?
    - If an element can be picked in an array, make sure that position < length of array.
- Are direct addresses to memory locations valid and correct?
    - Ensures that important information is not being overwritten.

☐ **Incorrect Calculation of Buffer Size**

- Do buffer size calculations include NULL pointers and take into account any types of overflows (integer, long, etc.)?

☐ **Incorrect Permission Assignment for Critical Resource**

- Are permissions are set to the lowest possible and only heightened for specific purposes?
    - DO NOT rely on users to read the documentation to modify permissions.
- Does the code account for the environment in which the software is running?

☐ **Authentication Bypass by Spoofing**

- Are IP verification, key exchanges, and filtering and access lists used during authentication?
- Are these checks occurring whenever an IP address is being taken as an argument or being used?
    - Spoofing attacks are when a malicious party impersonates a user or a device on a network in order to steal data, launch attacks on the network hosts, or bypass access controls.

☐ **Missing Encryption of Sensitive Data**

- Is all confidential or sensitive information encrypted in a strong and secure manner?

☐ **Loops**

- Are loops free from Logical errors?
    - Such as missing parentheses in the expression (a + b) / 2
- Are loops free from mathematical errors?
    - Basic math errors or incorrect incrementing / decrementing
- Are loops free from variable handling errors
    - Loop variables are initialized and modified correctly
- Are termination conditions defined to limit the number of times to try to achieve a result?
    - Like attempting to connect to a server when it is down.

☐ **Deadlock**

- Are at least one of the following Coffman conditions guaranteed to NOT be true for resource access?
    - Mutual exclusion: at least one resource is held in a non-shareable mode. Only one process can use the resource at any given instant of time.
    - Hold and wait or resource holding: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
    - No preemption: a resource can be released only voluntarily by the process holding it.
    - Circular wait: a process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, . . . PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on until PN is waiting for a resource held by P1."

☐ **Race Condition During Access to Alternate Channel**

- Are channels to communicate to users secure such that outside users cannot hijack into it?

☐ **Resource Management Errors**

- Do resources have limits to prevent exhaustion by users?
    - Limit threads, etc.
- Are objects never used after being freed from memory?
- Are all objects freed only once?

☐ **Use of Hard-coded Credentials**

- Are hard coded credentials hashed and placed in configuration files or databases with proper security?
- Are limits set on which entities can access features that require the hard-coded credentials?

☐ **Use of Potentially Dangerous Function**

- Is the use of dangerous functions avoided by using their safer equivalents?
    - gets , printf, memset, strcpy, etc.
- If not, are they being properly set up to avoid vulnerabilities?

☐ **Access of Memory Location after End of Buffer**

- Is all pointer arithmetic correct and accessing memory locations within the bounds of the buffer?
- Are the buffer's bounds well defined to ensure that memory is not being tampered with in unexpected ways?

☐ **Information Exposure**

- Are all areas of code that deal with sensitive information safely and correctly handling the information?
    - Make sure it is being encrypted, if necessary, and check for any outbound connections to unsafe areas of the code.

☐ **Integer Overflow or Wraparound**

- Is all arithmetic for integers, shorts, or any other numeric data type correct?
    - Specifically check for cases where user input is allowed or cases where one type is being converted to another (int to short for example).
- Are the proper types for values being used?
    - If expecting large numbers, use long instead of int, for example.

☐ **Unauthorized I/O Operations on a File (File Descriptor Leak)**

- Are all file descriptors closed before forking or executing a child process?

☐ **Temporary Files**

- Are sufficiently random names used to prevent guessing of temporary file names?
- Are files created or opened in a single operation to prevent race conditions?
- Are files created or opened such that they fail if the file already?

APPENDIX D: SAMPLE REQUIREMENTS

The below are sample requirements that can be leveraged for software security.

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The Program shall define policy and procedures to ensure that the developed or delivered systems do not embed unencrypted static authenticators in applications, access scripts, configuration files, nor store unencrypted static authenticators on function keys.		IA-5(7)
The system shall restrict the use of information inputs to the system and designated ground stations as defined in the applicable ICDs.		SC-23, SI-10,SI-10(5)
The system shall implement cryptography for the indicated uses using the indicated protocols, algorithms, and mechanisms, in accordance with applicable federal laws, Executive Orders, directives, policies, regulations, and standards: [NSA- certified or approved cryptography for protection of classified information, FIPS-validated cryptography for the provision of hashing] in accordance with applicable federal laws, Executive Orders, directives, policies, regulations, and standards.		IA-7, SC-13
The Program shall require the developer of the system, system component, or system services to demonstrate the use of a system development life cycle that includes [state-of-the-practice system/security engineering methods, software development methods, testing/evaluation/validation techniques, and quality control processes].	Examples of good security practices would be using defense-in-depth tactics across the board, least-privilege being implemented, two factor authentication everywhere possible, using DevSecOps, implementing and validating adherence to secure coding standards, performing static code analysis, component/origin analysis for open source, fuzzing/dynamic analysis with abuse cases, etc.	SA-3, SA-4(3)
The Program shall require subcontractors developing information system components or providing information system services (as appropriate) to demonstrate the use of a system development life cycle that includes [state-of-the-practice system/security engineering methods, software development methods, testing/evaluation/validation techniques, and quality control processes].	Select the particular subcontractors, software vendors, and manufacturers based on the criticality analysis performed for the PPP and the criticality of the components that they supply.	SA-3, SA-4(3)
The Program shall require the developer of the system, system component, or system service to deliver the system, component, or service with [Program-defined security configurations] implemented.	For the software, the defined security configuration could include to ensure the software does not contain a pre-defined list of Common Weakness Enumerations (CWEs) and/or CAT I/II Application STIGs.	SA-4(5)
The Program shall require the developer of the system, system component, or system service to use [Program-defined security configurations] as the default for any subsequent system, component, or service reinstallation or upgrade.		SA-4(5)
The Program shall review proposed changes to the system, assessing both mission and security impacts.		SA-10, CM-3(2)
The Program shall perform configuration management during system, component, or service during [design; development; implementation; operations].		SA-10

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The Program prohibits the use of binary or machine-executable code from sources with limited or no warranty and without the provision of source code.		SI-7(14)
The system shall prevent the installation of Flight Software without verification that the component has been digitally signed using a certificate that is recognized and approved by the Program.		CM-5(3)
The Program shall perform and document threat and vulnerability analyses of the as-built system, system components, or system services.		SA-11(2)
The Program shall use the threat and vulnerability analyses of the as-built system, system components, or system services to inform and direct subsequent testing/evaluation of the as-built system, component, or service.		SA-11(2)
The Program shall perform a manual code review of all flight code.		SA-11(4)
The Program shall conduct an Attack Surface Analysis and reduce attack surfaces to a level that presents a low level of compromise by an attacker.		SA-11(6), SA-15(5)
The Program shall use threat modeling and vulnerability analysis to inform the current development process using analysis from similar systems, components, or services where applicable.		SA-15(4), SA-15(8)
The Program shall create and implement a security assessment plan that includes: (1) The types of analyses, testing, evaluation, and reviews of [all] software and firmware components; (2) The degree of rigor to be applied to include abuse cases and/or penetration testing; and (3) The types of artifacts produced during those processes.	The security assessment plan should include evaluation of mission objectives in relation to the security of the mission. Assessments should not only be control based but also functional based to ensure mission is resilient against failures of controls.	SA-11, SA-11(5),CA-8
The Program shall verify that the scope of security testing/evaluation provides complete coverage of required security controls (to include abuse cases and penetration testing) at the depth of testing defined in the test documents.	* The frequency of testing should be driven by Program completion events and updates. * Examples of approaches are static analyses, dynamic analyses, binary analysis, or a hybrid of the three approaches	SA-11(5), SA-11(7),CA-8
The Program shall perform [Selection (one or more): unit; integration; system; regression] testing/evaluation at [Program-defined depth and coverage].	The depth needs to include functional testing as well as negative/abuse testing.	SA-11
The Program shall maintain evidence of the execution of the security assessment plan and the results of the security testing/evaluation.		SA-11, CA-8
The Program shall implement a verifiable flaw remediation process into the developmental and operational configuration management process.	The verifiable process should also include a cross reference to mission objectives and impact statements. Understanding the flaws discovered and how they correlate to mission objectives will aid in prioritization.	SA-11
The Program shall correct flaws identified during security testing/evaluation.	Flaws that impact the mission objectives should be prioritized.	SA-11

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The Program shall perform vulnerability analysis and risk assessment of [all systems and software].		SA-15(7), RA-5
The Program shall identify, report, and coordinate correction of cybersecurity-related information system flaws.		SI-2
The Program shall correct reported cybersecurity-related information system flaws, as requested.	* Although this requirement is stated to specifically apply to cybersecurity-related flaws, the Program office may choose to broaden it to all system flaws. * This requirement is allocated to the Program, as it is presumed, they have the greatest knowledge of the components of the system and when identified flaws apply.	SI-2
The Program shall test software and firmware updates related to flaw remediation for effectiveness and potential side effects on mission systems in a separate test environment before installation.	This requirement is focused on software and firmware flaws. If hardware flaw remediation is required, refine the requirement to make this clear.	SI-2, CM-3(2),CM-4(1)
The Program shall release updated versions of the mission information systems incorporating security-relevant software and firmware updates, after suitable regression testing, at a frequency no greater than [Program-defined frequency [90 days]].	On-orbit patching/upgrades may be necessary if vulnerabilities are discovered after launch. The system should have the ability to update software post-launch.	CM-3(2), CM-4(1)
The system shall be capable of removing flight software after updated versions have been installed.		SI-2(6)
The Program shall report identified systems or system components containing software affected by recently announced cybersecurity-related software flaws (and potential vulnerabilities resulting from those flaws) to [Program-defined officials] with cybersecurity responsibilities in accordance with organizational policy.		SI-2
The Program shall ensure that vulnerability scanning tools and techniques are employed that facilitate interoperability among tools and automate parts of the vulnerability management process by using standards for: (1) Enumerating platforms, custom software flaws, and improper configurations; (2) Formatting checklists and test procedures; and (3) Measuring vulnerability impact.	Component/Origin scanning looks for open-source libraries/software that may be included into the baseline and looks for known vulnerabilities and open source license violations.	RA-5
The Program shall create prioritized list of software weakness classes (e.g., Common Weakness Enumerations) to be used during static code analysis for prioritization of static analysis results.	The prioritized list of CWEs should be created considering operational environment, attack surface, etc. Results from the threat modeling and attack surface analysis should be used as inputs into the CWE prioritization process. There is also a CWSS (https://cwe.mitre.org/cwss/cwss_v1.0.1.html) process that can be used to prioritize CWEs. The prioritized list of CWEs can help with tools selection as well as you select tools based on their ability to detect certain high priority CWEs.	SA-11(1), SA-15(7)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The Program shall perform static source code analysis for [all available source code] looking for [Select one {Program-defined Top CWE List, SANS Top 25, OWASP Top 10}] weaknesses using no less than two static code analysis tools		SA-11(1), SA-15(7),RA-5
The Program shall perform component analysis (a.k.a. origin analysis) for developed or acquired software.	Component/Origin scanning looks for open-source libraries/software that may be included into the baseline and looks for known vulnerabilities and open source license violations.	SA-15(7), RA-5
The Program shall analyze vulnerability/weakness scan reports and results from security control assessments.		RA-5
The Program shall determine the vulnerabilities/weaknesses that require remediation, and coordinate the timeline for that remediation, in accordance with the analysis of the vulnerability scan report, the Program assessment of risk, and mission needs.		RA-5
The Program shall share information obtained from the vulnerability scanning process and security control assessments with [Program-defined personnel or roles] to help eliminate similar vulnerabilities in other systems (i.e., systemic weaknesses or deficiencies).		RA-5
The Program shall ensure that the vulnerability scanning tools (e.g., static analysis and/or component analysis tools) used include the capability to readily update the list of potential information system vulnerabilities to be scanned.		RA-5(1)
The Program shall ensure that the list of potential system vulnerabilities scanned is updated [prior to a new scan]		RA-5(2)
The Program shall define acceptable coding languages to be used by the software developer.		SA-15
The Program shall define acceptable secure coding standards for use by the developer.		SA-15
The Program shall have automated means to evaluate adherence to coding standards.		SA-15, SA-15(7), RA-5
The Program shall employ dynamic analysis (e.g., using simulation, penetration testing, fuzzing, etc.) to identify software/firmware weaknesses and vulnerabilities in developed and incorporated code (open source, commercial, or third-party developed code).	Fuzzing and/or dynamic analysis with abuse cases is important to flush out edge cases and how malicious actors could affect the system's software. Not all defects (i.e., buffer overflows, race conditions, and memory leaks) can be discovered statically and require execution of the software. This is where cyber testbeds (i.e., cyber ranges) are imperative as they provide an environment to maliciously attack components in a controlled environment to discover these undesirable conditions. Technology has improved to where digital twins for embedded systems are achievable, which provides an avenue for cyber testing that was often not performed due to perceived risk to the flight hardware.	SA-11(5), SA-11(8),CA-8

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The Program shall protect against supply chain threats to the system, system components, or system services by employing [institutional-defined security safeguards]	The chosen supply chain safeguards should demonstrably support a comprehensive, defense-in-breadth information security strategy. Safeguards should include protections for both hardware and software.	SA-12
The Program shall conduct a criticality analysis to identify mission critical functions and critical components and reduce the vulnerability of such functions and components through secure system design.	During SCRM, criticality analysis will aid in determining supply chain risk. For mission critical functions/components, extra scrutiny must be applied to ensure supply chain is secured.	SA-12, SA-14,SA-15(3),CP-2(8)
The Program shall request threat analysis of suppliers of critical components and manage access to and control of threat analysis products containing U.S. person information.		SA-12
The Program shall employ the [Program-defined] approaches for the purchase of the system, system components, or system services from suppliers.	This could include tailored acquisition strategies, contract tools, and procurement methods.	SA-12(1)
The Program shall maintain documentation tracing the strategies, tools, and methods implemented to the Program-defined strategies, tools, and methods as a means to mitigate supply chain risk .	Examples include: (1) Transferring a portion of the risk to the developer or supplier through the use of contract language and incentives; (2) Using contract language that requires the implementation of SCRM throughout the system lifecycle in applicable contracts and other acquisition and assistance instruments (grants, cooperative agreements, Cooperative Research and Development Agreements (CRADAs), and other transactions). Within the DoD some examples include: (a) Language outlined in the Defense Acquisition Guidebook section 13.13. Contracting; (b) Language requiring the use of protected mechanisms to deliver elements and data about elements, processes, and delivery mechanisms; (c) Language that articulates that requirement flow down supply chain tiers to sub-prime suppliers. (3) Incentives for suppliers that: (a) Implement required security safeguards and SCRM best practices; (b) Promote transparency into their organizational processes and security practices; (c) Provide additional vetting of the processes and security practices of subordinate suppliers, critical information system components, and services; and (d) Implement contract to reduce SC risk down the contract stack. (4) Gaining insight into supplier security practices; (5) Using contract language and incentives to enable more robust risk management later in the lifecycle; (6) Using a centralized intermediary or “Blind Buy” approaches to	SA-12(1)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
	acquire element(s) to hide actual usage locations from an untrustworthy supplier or adversary;	
The Program shall employ [Selection (one or more): independent third-party analysis, Program penetration testing, independent third-party penetration testing] of [Program-defined supply chain elements, processes, and actors] associated with the system, system components, or system services.		SA-12(11)
The Program shall perform penetration testing/analysis: (1) On potential system elements before accepting the system; (2) As a realistic simulation of the active adversary's known adversary tactics, techniques, procedures (TTPs), and tools; and (3) Throughout the lifecycle on physical and logical systems, elements, and processes.	Penetration testing should be performed throughout the lifecycle on physical and logical systems, elements, and processes including: (1) Hardware, software, and firmware development processes; (2) Shipping/handling procedures; (3) Personnel and physical security programs; (4) Configuration management tools/measures to maintain provenance; and (5) Any other programs, processes, or procedures associated with the production/distribution of supply chain elements.	SA-11(5)
The Program shall employ [Program-defined] techniques to limit harm from potential adversaries identifying and targeting the Program supply chain.	Examples of security safeguards that the organization should consider implementing to limit the harm from potential adversaries targeting the organizational supply chain, are: (1) Using trusted physical delivery mechanisms that do not permit access to the element during delivery (ship via a protected carrier, use cleared/official couriers, or a diplomatic pouch); (2) Using trusted electronic delivery of products and services (require downloading from approved, verification-enhanced sites); (3) Avoiding the purchase of custom configurations, where feasible; (4) Using procurement carve outs (i.e., exclusions to commitments or obligations), where feasible; (5) Using defensive design approaches; (6) Employing system OPSEC principles; (7) Employing a diverse set of suppliers; (8) Employing approved vendor lists with standing reputations	SA-12(5), SC-38

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
	in industry; (9) Using a centralized intermediary and "Blind Buy" approaches to acquire element(s) to hide actual usage locations from an untrustworthy supplier or adversary Employing inventory management policies and processes; (10) Using flexible agreements during each acquisition and procurement phase so that it is possible to meet emerging needs or requirements to address supply chain risk without requiring complete revision or re-competition of an acquisition or procurement; (11) Using international, national, commercial or government standards to increase potential supply base; (12) Limiting the disclosure of information that can become publicly available; and (13) Minimizing the time between purchase decisions and required delivery.	
The Program shall use all-source intelligence analysis of suppliers and potential suppliers of the information system, system components, or system services to inform engineering, acquisition, and risk management decisions.	* The Program should also consider sub suppliers and potential sub suppliers. * All-source intelligence of suppliers that the organization may use includes: (1) Defense Intelligence Agency (DIA) Threat Assessment Center (TAC), the enterprise focal point for supplier threat assessments for the DoD acquisition community risks; (2) Other U.S. Government resources including: (a) Government Industry Data Exchange Program (GIDEP) – Database where government and industry can record issues with suppliers, including counterfeits; and (b) System for Award Management (SAM) – Database of companies that are barred from doing business with the US Government.	SA-12(8)
The Program (and Prime Contractor) shall conduct a supplier review prior to entering into a contractual agreement with a contractor (or sub-contractor) to acquire systems, system components, or system services.		SA-12(2)
The Program shall maintain a list of suppliers and potential suppliers used, and the products that they supply to include software.	Ideally you have diversification with suppliers	PL-8(2)
The Program shall employ [Program-defined Operations Security (OPSEC) safeguards] to protect supply chain-related information for the system, system components, or system services.	OPSEC safeguards may include: (1) Limiting the disclosure of information needed to design, develop, test, produce, deliver, and support the element for example, supplier identities, supplier processes, potential suppliers, security requirements, design specifications, testing and evaluation result, and system/component configurations, including	SA-12(9), SC-38,CP-2(8)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
	the use of direct shipping, blind buys, etc.; (2) Extending supply chain awareness, education, and training for suppliers, intermediate users, and end users; (3) Extending the range of OPSEC tactics, techniques, and procedures to potential suppliers, contracted suppliers, or sub-prime contractor tier of suppliers; and (4) Using centralized support and maintenance services to minimize direct interactions between end users and original suppliers.	
The Program shall develop and implement anti-counterfeit policy and procedures designed to detect and prevent counterfeit components from entering the information system, including support tamper resistance and provide a level of protection against the introduction of malicious code or hardware.		SA-19
The Program shall develop and implement anti-counterfeit policy and procedures, in coordination with the [CIO], that is demonstrably consistent with the anti-counterfeit policy defined by the Program office.		SA-19
The system shall maintain the confidentiality and integrity of information during preparation for transmission and during reception.		SC-8(2)
The system shall protect the confidentiality and integrity of the following information using cryptography while it is at rest: [all information].		SC-28, SC-28(1),SI-7(6)
The Program shall enable integrity verification of software and firmware components.		SA-10(1), SI-7
The system shall perform an integrity check of [Program-defined software, firmware, and information] at startup; at [Program-defined transitional states or security-relevant events]		SI-7(1)
The Program shall define and document the transitional state or security-relevant events when the system will perform integrity checks on software, firmware and information.		SI-7(1)
The Program shall employ automated tools that provide notification to [Program-defined personnel] upon discovering discrepancies during integrity verification.		SI-7(2)
The Program shall define the security safeguards that are to be employed when integrity violations are discovered.		SI-7(5)
The Program shall ensure that the contractors/developers have all ASICs designed, developed, manufactured, packaged, and tested by suppliers with a Defense Microelectronics Activity (DMEA) Trust accreditation.		SA-12, SA-12(1)
The [software subsystem] shall operate securely in off-nominal power conditions, including loss of power and spurious power transients.		SI-17
The [software subsystem] shall identify and reject commands received out-of-sequence when the out-of-sequence commands can cause a hazard/failure or degrade the control of a hazard or mission.		SI-10

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The [software subsystem] shall detect and recover from detected memory errors or transitions to a known cyber-safe state.		SI-17
The [software subsystem] shall recover to a known cyber-safe state when an anomaly is detected.		SI-17
The [software subsystem] shall accept [Program defined hazardous] commands only when prerequisite checks are satisfied.		SI-10
The [software subsystem] shall safely transition between all predefined, known states.		SI-17
The [software subsystem] shall discriminate between valid and invalid input into the software and rejects invalid input.		SI-10, SI-10(3)
The [software subsystem] shall properly handle spurious input and missing data.		SI-10, SI-10(3)
The system shall have failure tolerance on sensors used by software to make mission-critical decisions.		SI-17
The [software subsystem] shall provide two independent and unique command messages to deactivate a fault tolerant capability for a critical or catastrophic hazard.		AC-3(2)
The [software subsystem] shall provide at least one independent command for each operator-initiated action used to shut down a function leading to or reducing the control of a hazard.		SI-10(5)
The [software subsystem] shall provide non-identical methods, or functionally independent methods, for commanding a mission critical function when the software is the sole control of that function.		AC-3(2)
The [software subsystem] shall provide independent mission/cyber critical threads such that any one credible event will not corrupt another mission/cyber critical thread.		SC-3
The system’s mission/cyber critical commands shall require to be "complex" and/or diverse from other commands so that a single bit flip could not transform a benign command into a hazardous command.		SI-10(5)
The [software subsystem] shall perform prerequisite checks for the execution of hazardous commands.		SI-10
The [software subsystem] shall validate a functionally independent parameter prior to the issuance of any sequence that could remove an inhibit or perform a hazardous action.		SI-10(3)
The [Program-defined security policy] shall state that information should not be allowed to flow between partitioned applications unless explicitly permitted by the Program's security policy.		AC-4
The Program shall identify the key system components or capabilities that require isolation through physical or logical means.		SC-3
The system shall enforce approved authorizations for controlling the flow of information within the system and between interconnected systems based on the [Program defined security policy] that information does not leave the system boundary unless it is encrypted.		AC-4

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The system shall, when transferring information between different security domains, implements the following security policy filters that require fully enumerated formats that restrict data structure and content: connectors and semaphores implemented in the RTOS.		AC-4(14)
The system shall use protected processing domains to enforce the policy that information does not leave the system boundary unless it is encrypted as a basis for flow control decisions.		AC-4(2)
The system shall isolate [Program-defined] mission critical functionality from non-mission critical functionality by means of an isolation boundary (implemented via partitions) that controls access to and protects the integrity of, the hardware, software, and firmware that provides that functionality.	* Examine the isolation between mission critical and non-mission critical functionality for each individual information system component. Include architectural considerations in the examination, including isolation derived from using distinct components for mission critical and non-mission critical functionality. This would include having multiple 1553 buses for example to segregate C&DH/TT&C with payload operations. * Methods to separate the mission/cyber critical software from software that is not critical, such as partitioning, may be used (i.e., ARINC 653). If such software methods are used to separate the code and are verified, then the software used in the isolation method is mission/cyber critical, and the rest of the software is not mission/cyber critical.	SC-3
The system data within partitioned applications shall not be read or modified by other applications/partitions.		SC-4, SC-6
The system shall employ the principle of least privilege, allowing only authorized accesses processes which are necessary to accomplish assigned tasks in accordance with system functions.		AC-6
The system shall maintain a separate execution domain for each executing process.		SC-7(21), SC-39
The system shall ensure that processes reusing a shared system resource (e.g., registers, main memory, secondary storage) do not have access to information (including encrypted representations of information) previously stored in that resource during a prior use by a process after formal release of that resource back to the system or reuse.		SC-4
The system shall prevent unauthorized and unintended information transfer via shared system resources.		SC-4
The system software must not be able to tamper with the security policy or its enforcement mechanisms.		SC-3
The system protects the availability of resources by allocating [Program-defined] resources based on [priority and/or quota].		SC-6
The trusted boot/RoT shall be a separate compute engine controlling the trusted computing platform cryptographic processor.		SI-7(9)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The trusted boot/RoT computing module shall be implemented on radiation tolerant burn-in (non-programmable) equipment.		SI-7(9)
The system boot firmware must verify a trust chain that extends through the hardware root of trust, boot loader, boot configuration file, and operating system image, in that order.		SI-7(9)
The system boot firmware must enter a recovery routine upon failing to verify signed data in the trust chain, and not execute or trust that signed data.		SI-7(9)
The system shall allocate enough boot ROM memory for secure boot firmware execution.		SI-7(9)
The system shall allocate enough SRAM memory for secure boot firmware execution.		SI-7(9)
The system secure boot mechanism shall be Commercial National Security Algorithm Suite (CNSA) compliant.		SI-7(9)
The system shall support the algorithmic construct Elliptic Curve Digital Signature Algorithm (ECDSA) NIST P-384 + SHA-38		SI-7(9)
The system hardware root of trust must be an ECDSA NIST P-384 public key.		SI-7(9)
The system hardware root of trust must be loadable only once, post-purchase.		SI-7(9)
The system boot firmware must validate the boot loader, boot configuration file, and operating system image, in that order, against their respective signatures.		SI-7(9)
The Program shall perform static binary analysis of all firmware that is utilized on the spacecraft.	Many commercial products/parts are utilized within the system and should be analyzed for security weaknesses. Blindly accepting the firmware is free of weakness is unacceptable for high assurance missions.	SA-11, RA-5
The Program shall define/maintain an approved operating system list for use on spacecraft.	The operating system is extremely important to security and availability of the system, therefore should receive high levels of assurance that it operates as intended and free of critical weaknesses/vulnerabilities.	CM-7(5)
The system's operating system, if COTS or FOSS, shall be selected from a [Program-defined] accepted list.		SI-7(14), CM-7(5)
The system shall retain the capability to update/upgrade operating systems while in operations.	The operating system updates should be performed using multi-factor authorization and should only be performed when risk of compromise/exploitation of identified vulnerability outweighs the risk of not performing the update.	SA-4(5)
The Program shall define acceptable secure communication protocols available for use within the mission in accordance with applicable federal laws, Executive Orders, directives, policies, regulations, and standards.		SA-4(9)
The system shall only use [Program-defined] communication protocols within the mission.		SA-4(9)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
The system shall protect the confidentiality and integrity of the following transmitted information: "all transmitted information".		SC-8
The system shall implement cryptographic mechanisms to prevent unauthorized disclosure of, and detect changes to, information during transmission unless otherwise protected by no alternative physical safeguards.		SC-8(1), SI-7(6)
The system shall maintain the confidentiality and integrity of information during preparation for transmission and during reception.		SC-8(2)
The system shall implement cryptographic mechanisms to protect message externals unless otherwise protected by no alternative physical safeguards.		SC-8(3)
The system shall generate error messages that provide information necessary for corrective actions without revealing information that could be exploited by adversaries.		SI-11
The Program shall conduct an assessment of risk, including the likelihood and magnitude of harm, from the unauthorized access, use, disclosure, disruption, modification, or destruction of the system and the information it processes, stores, or transmits.		RA-3
The Program's risk assessment shall include the full end to end communication pathway from the ground to the spacecraft.		RA-3
The Program shall document risk assessment results in [risk assessment report].		RA-3
The Program shall review risk assessment results [At least annually if not otherwise defined in formal organizational policy].		RA-3
The Program shall update the risk assessment [At least annually if not otherwise defined in formal institutional policy] or whenever there are significant changes to the information system or environment of operation (including the identification of new threats and vulnerabilities), or other conditions that may impact the security state of the system.		RA-3
The Program shall coordinate penetration testing on [program-defined mission critical system components (hardware and/or software)].		CA-8
The system shall be designed and configured so that [Program-defined encrypted communications traffic and data] is visible to system monitoring tools.		SI-4(10)
The system shall integrate cyber related detection and responses with existing fault management capabilities to ensure tight integration between traditional fault management and cyber intrusion detection and prevention.		AU-6(4), SI-4(16)
The system shall provide an alert immediately to [at a minimum the mission director, administrators, and security officers] when the following failure events occur: [minimally but not limited to auditing software/hardware errors; failures in the audit capturing mechanisms; and audit storage capacity reaching 95%, 99%, and 100%] of allocated capacity.		AU-5(2)
The Program shall document and design a security architecture using a defense-in-depth approach that allocates the Program defined safeguards to the indicated locations and layers: [Examples		PL-8, PL-8(1)

Requirement Text	Rationale / Additional Guidance / Notes	NIST 800-53 rev4 Control Mapping (if exists)
include operating system abstractions and hardware mechanisms to the separate processors in the system, internal components, and the software].		
The Program shall implement a security architecture and design that provides the required security functionality, allocates security controls among physical and logical components, and integrates individual security functions, mechanisms, and processes together to provide required security capabilities and a unified approach to protection.		SA-2, SA-8

Appendix E: Glossary of Terms

The source of this information is from NASA's Software Assurance Research Program (SARP) initiative on secure software development.

Abstraction Library/Layer

A way of hiding the implementation details of a particular set of functionalities. Usually implemented using existing functions and features (sometimes third-party libraries or frameworks) that are proven to work and are well tested.

Attack Modeling

Attack Modeling examines the application from an attacker's perspective using known attack patterns to see what the system is vulnerable to. Attack Modeling differs from Threat Modeling in that Attack Modeling identifies specific attacks to the system that may be successful, while Threat Modeling identifies vulnerabilities in the system that may be subject to attack.

Authentication

The process of confirming the identity of a user or other external entity prior to granting them access to the system. Authentication is performed by verifying the credentials provided by the external entity, which may be single factor (username and password), two factor (a hardware token and pin, a security token code and pin, biometric data and pin), or multi-factor (more than two of any of the previously mentioned credentials).

Black Box Testing

A method of software testing that examines the functionality of an application without visibility into its internal structure or design. Only the inputs to the tested item and the expected outputs are known. This method of testing can be applied to virtually every level of software testing: unit, integration, system, and acceptance.

Checksum

A data value representing the sum of the correct digits of a data file or message. The value is typically calculated by the sender before transmission and is sent with the transmitted data file or message. The receiver of the information then calculates the checksum of the received file or message and compares it to the sender's checksum value. If the values do not match, then the data was either corrupted or tampered with during transmission and should not be used.

Communication Protocol

A formal description of digital message formats and rules. They are required to exchange messages between or inside computing systems. Communication protocols define properties of

transmission such as packet size, transmission speed, synchronization techniques, routing, and error correction types. Some popular protocols include File Transfer Protocol (FTP), TCP/IP, User Datagram Protocol (UDP), Hypertext Transfer Protocol (HTTP), and Simple Mail Transfer Protocol (SMTP).

Cryptographic Protocol

An abstract or concrete protocol that performs a security-related function and applies cryptographic methods. The protocol assures confidentiality, message integrity, and sometimes even anonymity. Some common protocols include RSA and AES encryption protocols.

Deadlock

A situation where two computer processes sharing the same resource effectively prevent each other from accessing the resource. The result is that both processes cease to function, waiting on the locked resource. Once deadlock occurs, all other processes attempting to access this same resource will also be blocked. To resolve the deadlock, one of the originally locking processes must be aborted.

Encryption

The translation of data using an encryption algorithm and encryption key into cyphertext that is only accessible by authorized parties with access to the correct key. Proper encryption makes data impossible to read without the key or password that allows for decryption.

Format String

An ASCII Z string that contains text and format parameters to define the desired output format within a format function, such as printf or fprintf. The format parameter defines the type of conversion performed by the format function and is specified as %X, where X represents the desired output format.

Fuzzing

Fuzzing is a black box software testing technique that provides invalid, unexpected, or random data as inputs to the application to discover issues or security weaknesses. There are three principal fuzzing techniques: Random fuzzing, Template fuzzing (also known as mutation fuzzing or block fuzzing) and Generational fuzzing (also known as model, RFC or standards-based fuzzing).

- Random fuzzing - generates random data and is the easiest way of fuzzing. However, random data often lacks the form and structure that is needed for an effective fuzz.
- Template fuzzing - uses a good, known template (content from file, traffic capture, PC/RPC - API call, etc.) as a starting point for fuzzing. Basically, a script of data inputs

is defined and replayed over and over, but with changes to the data. The quality of the template fuzzing heavily depends on the effectiveness of the selected templates.

- Generational fuzzing - the best kind of fuzzing where tests are created with a complete understanding of the fuzzed content or protocol and its specification. For example, an HTTP fuzzer has been implemented using the HTTP specifications and it knows every type of message type, every field of every message, and rules about how messages are exchanged.

Penetration Testing

Penetration testing, informally called pen testing, is an attack on a computer system that looks for security weaknesses, potentially gaining access to the computer's features and data. The process typically identifies the target systems and a particular goal - then reviews available information and undertakes various means to attain the goal. A penetration test target may be a white box (which provides background and system information) or black box (which provides only basic or no information except the company name). A penetration test can help determine whether a system is vulnerable to attack, if the defenses were sufficient, and which defenses (if any) the test defeated.

Stress Testing

A software testing activity that determines the robustness of software by intentionally exceeding the limits of normal operation and monitoring the resulting behavior.

Threat Modelling

A procedure for optimizing security by identifying objectives and vulnerabilities and defining countermeasures to prevent or mitigate the effects of threats to the system. Essentially, it is examining the application and its environment from a security perspective to identify weaknesses. Identify the threats and potential mitigations of those threats to a system by:

- decomposing the application
- defining and classifying assets
- exploring potential vulnerabilities
- exploring potential threats
- creating mitigation strategies

Throttling Mechanism

A method to regulate the rate or volume of throughput to an application or resource. Its goal is to optimize available system resources for active processes and prevent unsustainable consumption or resource exhaustion. Throttling may be implemented by restricting the number of threads or

connections, using load leveling, deferring lower priority operations, or degrading the performance of selected operations.

XSS (Cross-Site-Scripting)

A type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.

BIBLIOGRAPHY

(n.d.). Retrieved from https://www.whitesourcesoftware.com/most-secure-programming-languages/

(n.d.). Retrieved from https://cuckoosandbox.org/

(n.d.). Retrieved from https://cyber.gc.ca/en/assemblyline

(n.d.). Retrieved from https://www.virustotal.com/gui/

Allen, F. E., & Cocke, J. (1976). A Program Data Flow Analysis Procedure. *Communications of the ACM, 19*(3).

Assistant Secretary of Defense for Networks and Information Integration / Department of Defense Chief Innovation Officer. (2015). *Information Assurance Workforce Improvement Program.* Department of Defense.

Avgerinos, T., Brumley, D., Davis, J., Goulden, R., Nighswander, T., Rebert, A., & Williamson, N. (2018). The Mayhem Cyber Reasoning System. *IEEE Security and Privacy, 16*(2), 52-60.

Bass, L., & Clements, P. K. (2003). *Software Architecture in Practice, 2nd Edition.* New York, NY: Addison-Wesley.

Bird, J., & Allen, B. (2018). *Introduction to Secure DevOps.* North Bethesda: SANS Institute.

Black Duck Software. (2018, September). Retrieved from Black Duck Software: https://www.blackducksoftware.com/

Black, P. E., & Bojanova, I. (2016). Defeating Buffer Overflow: A Trivial but Dangerous Bug. *IT Professional, 18*(6), pp. 58-61. doi:10.1109/mitp.2016.117

Burnett, M. (2004, 12 20). *Security Holes That Run Deep*. (SecurityFocus) Retrieved from https://www.securityfocus.com/columnists/285

Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008).*

Carnegie Mellon University. (n.d.). *ENV06-J. Production code must not contain debugging entry points*. Retrieved from https://wiki.sei.cmu.edu/confluence/display/java/ENV06-J.+Production+code+must+not+contain+debugging+entry+points

CERT. (2016). *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition).* CERT. Retrieved from https://resources.sei.cmu.edu/forms/secure-coding-form.cfm

CERT. (2017). *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition).* Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=494932

Conway, M. (1968). How Do Committees Invent? *Datamation Magazine*.

Corporation, M. (n.d.). *CWE-134: Use of Externally-Controlled Format String*. Retrieved from https://cwe.mitre.org/data/definitions/134.html

Corporation, M. (n.d.). *CWE-170: Improper Null Termination*. Retrieved from https://cwe.mitre.org/data/definitions/170.html

Corporation, M. (n.d.). *CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')*. Retrieved from https://cwe.mitre.org/data/definitions/22.html

Corporation, M. (n.d.). *CWE-243: Creation of chroot Jail Without Changing Working Directory*. Retrieved from https://cwe.mitre.org/data/definitions/243.html

Corporation, M. (n.d.). *CWE-465: Pointer Issues*. Retrieved from https://cwe.mitre.org/data/definitions/465.html

Corporation, M. (n.d.). *CWE-732: Incorrect Permission Assignment for Critical Resource*. Retrieved from https://cwe.mitre.org/data/definitions/732.html

Corporation, M. (n.d.). *CWE-770: Allocation of Resources Without Limits or Throttling*. Retrieved from https://cwe.mitre.org/data/definitions/770.html

Corporation, M. (n.d.). *CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')*. Retrieved from https://cwe.mitre.org/data/definitions/835.html

Cousout, P. (2005). *MIT Course 16.399: « Abstract Interpretation » Home Page*. Retrieved from Massachusetts Institute of Technology: http://web.mit.edu/16.399/www/

Defense Advanced Research Projects Agency. (n.d.). *Cyber Grand Challenge (CGC)*. Retrieved October 23, 2018, from https://www.darpa.mil/program/cyber-grand-challenge

Defense Information Systems Agency. (2018, September). *Information Assurance Support Environment*. Retrieved from https://iase.disa.mil/Pages/index.aspx

Defense Information Systems Agency. (n.d.). *Application Security and Development STIG*. Retrieved from https://iasecontent.disa.mil/stigs/zip/U_ASD_V4R7_STIG.zip

Ferguson, J. (2007). Understanding the heap and breaking it: A case study of the heap as a persistent data structure through non-traditional exploitation techniques. *BlackHat.* Las Vegas, NV.

Fernandez, E., Astudillo, H., & Pedraza-García, G. (2014). *Revisiting architectural tactics for security.* Guayaquil: LACCEI.

ForAllSecure. (2018, September 28). *ForAllSecure homepage*. Retrieved from https://forallsecure.com/

Gamma E., H. R. (2000). *Design Patterns, Elements of Reusable Object Oriented Software.* New York: Addison-Wesley.

Godefroid, P. (2010). From Blackbox Fuzzing to Whitebox Fuzzing towards Verification. *International Symposium on Software Testing and Analysis.*

Godefroid, P., Levin, M. Y., & Molnar, D. (2012, 01 11). SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*.

Goodin, D. (2017, September 16). *Devs unknowingly use "malicious" modules snuck into official Python repositories*. (Ars Technica) Retrieved October 17, 2018, from https://arstechnica.com/information-technology/2017/09/devs-unknowingly-use-malicious-modules-put-into-official-python-repository/

Google. (n.d.). *AddressSanitizer*. Retrieved from https://github.com/google/sanitizers/wiki/AddressSanitizer

Google. (n.d.). *ThreadSanitizerAboutRaces*. Retrieved from https://github.com/google/sanitizers/wiki/ThreadSanitizerAboutRaces

Halfond, W. G., Viegas, J., & Orso, A. (2006). A Classification of SQL Injection Attacks. *IEEE ISSE*.

Hauser, J. R. (1996). Handling Floating-Point Exceptions in Numeric. *ACM Transactions on Programming Languages and Systems, 18*(2), 139-174.

Householder, A. D. (2018, March 9). *CERT BFF - Basic Fuzzing Framework*. Retrieved from https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework

Institute of Elecrical and Electronics Engineers. (1990). *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std. 610.12-1990).* New York, NY: IEEE.

Ivanti software. (n.d.). *HEAT Software home page*. Retrieved October 17, 2018, from https://www.ivanti.com/company/history/heat-software

Joint Task Force Transformation Initiative. (2018). *NIST Special Publication 800-53 rev 4 Security and Privacy Controls for Federal Information Systems and Organizations.* National Institute of STandards and Technology. Retrieved from https://nvd.nist.gov/800-53

Lapham, M. A., & Carol, W. (2006). *Sustaining Software-Intensive Systems.* Carnegie Mellon University.

lcamtuf. (2017, November 4). *american fuzzy lop*. Retrieved from GitHub: https://github.com/mirrorer/afl

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F., & Svoboda, D. (2011). *The CERT Oracle Secure Coding Standard for Java.* CERT.

Long, F., Mohindra, D., Seacord, R. C., Sutherland, D. F., & Svoboda, D. (2013). *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs.*

Manmoud, S. K., Alfonse, M., Roushdy, M. I., & M.Salem, A.-B. (2017). A comparative analysis of Cross Site Scripting (XSS) detecting and defensive techniques. *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, (p. 36).

Meyer, B. (2017, May). Ending Null Pointer Crashes. *Communications of the ACM, 60*(5), 8.

MITRE Corporation. (2018, September). *Common Weakness Enumeration*. Retrieved from https://cwe.mitre.org/

MITRE Corporation. (n.d.). *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer* . Retrieved from https://cwe.mitre.org/data/definitions/119.html

MITRE Corporation. (n.d.). *CWE-195: Signed to Unsigned Conversion Error*. Retrieved from https://cwe.mitre.org/data/definitions/195.html

MITRE Corporation. (n.d.). *CWE-20: Improper Input Validation*. Retrieved from https://cwe.mitre.org/data/definitions/20.html

MITRE Corporation. (n.d.). *CWE-681: Incorrect Conversion between Numeric Types* . Retrieved from https://cwe.mitre.org/data/definitions/681.html

MITRE Corporation. (n.d.). *CWE-682: Incorrect Calculation*. Retrieved from https://cwe.mitre.org/data/definitions/682.html

MITRE Corporation. (n.d.). *CWE-798: Use of Hard-coded Credentials*. Retrieved from https://cwe.mitre.org/data/definitions/798.html

Motor Industry Research Association (MISRA). (2013). *Guidelines for the Use of the C Language in Critical Systems.*

mstfcam. (2015, May 19). *Password Complexity versus Password Entropy*. Retrieved October 17, 2018, from https://blogs.technet.microsoft.com/msftcam/2015/05/19/password-complexity-versus-password-entropy/

Murphy, N. R., Petoff, J., Jones, C., & Beyer, B. (2016). *Site Reliability Engineering.* Sebastopol: O'Reilly Media, Inc.

National Institute of Standards and Technology. (2017). *NIST Special Publication 800-63B: Digital Identity Guidelines -- Authentication and Lifecycle Management.* Gaithersburg: National Institute of Standards and Technology.

National Vulnerability Database. (n.d.). Retrieved from https://nvd.nist.gov

National Vulnerability Database. (2014). *CVE-2014-0160*. Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

National Vulnerability Database. (2014, 09 09). *CVE-2014-6277*. Retrieved from National Vulnerability Database: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6277
National Vulnerability Database. (2017, 08 21). *CVE-2017-12983*. Retrieved from National Vulnerability Database: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12983
Newton, S., & Aschbacher, N. (2018). *The Challenge of Using C in Safety-Critical Applications.*
Neystadt, J. (2008, February). *Automated Penetration Testing with White-Box Fuzzing*. Retrieved from MSDN: https://msdn.microsoft.com/en-us/library/cc162782.aspx
NTIA. (n.d.). Retrieved from https://www.ntia.gov/files/ntia/publications/sbom_overview_20200818.pdf
Obiltschnig, G. (n.d.). Retrieved from https://www.appinf.com/download/SafetyCriticalC++.pdf
Offensive Security. (2018, September). *Offensive Security Home Page*. Retrieved from https://www.offensive-security.com/
Open Web Application Security Project (OWASP). (2017). *OWASP Top 10 - 2017.* OWASP. Retrieved from https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
Oracle. (2017). *Secure Coding Guidelines for Java SE.* Retrieved from https://www.oracle.com/technetwork/java/seccodeguide-139067.html
OWASP. (n.d.). *Double Free*. Retrieved from https://www.owasp.org/index.php/Double_Free
OWASP. (n.d.). *OWASP Dependency-Check*. Retrieved from https://www.owasp.org/index.php/OWASP_Dependency_Check
radare2. (2018, September). *radare2: unix-like reverse engineering framework and commandline tools security*. Retrieved from GitHub: https://github.com/radare/radare2
Reveliotis, S., & Fei, Z. (2017, October). Robust Deadlock Avoidance for Sequential Resource Allocation Systems with Resource Outages. *IEEE Transactions on Automation Science and Engineering, 14*(4), pp 1695-1711.
SANS Institute. (2018, September). *SANS Home Page*. Retrieved from https://www.sans.org
Scholte, T., Balzarotti, D., & Kirda, E. (2012). Have things changed now? An empirical study on input. *Computers & Security, 31*(3), 344-356.
Seacord, R. (n.d.). *Top 10 Secure Coding Practices*. (SEI CERT) Retrieved from https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices
Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., . . . Vigna, G. (2016). (State of) The Art of War: Offensive Techniques in Binary Analysis. *IEEE Symposium on Security and Privacy*.
Tanium Software. (n.d.). *Tanium Homepage*. Retrieved October 17, 2018, from https://www.tanium.com/
U.S. Air Force. (2015). *T.O. 00-35D-54 USAF Deficiency Reporting, Investigation, and Resolution.*
UC Santa Barbara. (2018, September). *angr binary analysis framework*. Retrieved from GitHub: https://github.com/angr
Unell, A., Deifik, J. S., Dietrich, S. M., & Haghverdian, G. (2018). *TOR-2018-01105 Software Assurance Pipeline Recommendations for the Space Defense Task Force.* El Segundo, CA: The Aerospace Corporation.
Valgrind Developers. (n.d.). *The Valgrind Quick Start Guide*. Retrieved from http://valgrind.org/docs/manual/quick-start.html

WhiteSource. (n.d.). Retrieved from https://www.whitesourcesoftware.com/most-secure-programming-languages/

zardus. (2018, 09 11). *preeny*. Retrieved from GitHub: https://github.com/zardus/preeny