# Supply Chain Threats - Software

White Paper

Authors:

Matthew Areno, PhD

Antonio Martin

July 2021

Version 1.0

# 1 Contents

# 2 Introduction

Supply chain attacks have seen significant increases both in quantity and severity over the last few years. These attacks have resulted in varying level of impacts to companies, from minimal to catastrophic. Such attacks have targeted both hardware and software products as attackers attempt to leverage any avenue possible to disrupt or compromise the integrity of these products. Protection of the software supply chain is critical to both the validation of software solutions and the prevention of malicious alterations to otherwise legitimate code.

The purpose of this document is to provide an outline of the software supply chain and the most critical attack points that need to be considered in order to mitigate associated attacks. This document should be used as a reference for organizations to assess their own mitigations and generate a prioritized plan for mitigating as many issues as possible. The attacks listed herein are extensive but not all-encompassing. Supply chain has an ever-increasing landscape of potential attacks. As such, this document, and subsequent assessments, require frequent reanalysis to remain up to date.

## 2.1 Acronyms

| Term | Description |
|---|---|
| **BU** | Business Unit |
| **DCBT** | Design->Code->Build->Test |
| **EOL** | End of Life |
| **HSM** | Hardware Security Module |
| **IDE** | Integrated Development Environment |
| **MITM** | Man-in-the-Middle |
| **SDK** | Software Development Kit |
| **SDLC** | Software Development Life Cycle |
| **TOCTOU** | Time-of-Check,Time-of-Use |
| **2FA** | Two Factor Authentication |

# 3 Software Supply Chain Overview

The software supply chain generally follows a flow referred to as the Software Development Life Cycle (SDLC). Unfortunately, there is no industry-defined version of the SDLC. There are a number of different variances of the life cycle that are most commonly based on the toolset or methodology being used, such as Agile or Waterfall. Each variance has its pros and cons, along with specific technologies and deployment strategies for which it works best.

This document will not provide a comprehensive review of all such variances, but will instead present a generic template for the SDLC that should then be mapped onto whatever approach is being used by the BU. A best guess of potential mappings will be provided but should be validated by each BU before proceeding.

## 3.1 Software Development Life Cycle

For the purposes of this document, the SDLC is a multi-stage process consisting of seven stages. Although there is a logical progress between stages, many stages, or even sequences of stages, may be repeated multiple times in an iterative fashion. The progressions and iterations represent the primary distinction between the various versions of SDLC. The stages and progression used in this document is as follows:

1. Concept
2. Design
3. Implement
4. Integrate
5. Build
6. Test
7. Deployment

The progression of these stages is illustrated below in Figure 1. As can the seen, with the exception of the Concept and Deployment stages, all other stages overlap with one another. A single software package or product is often composed of a large number of independent modules that are all integrated together. The modules may be built separately or may come from a third-party provider, hence the reason the Integrate and Build stages may start at the same time.

At some point, these modules are then integrated into the overall solution and built into the final product. Testing is likely also conducted iteratively as part of each modules' development. Once all integration is completed, final testing is still needed for the end product. Upon completion of testing, the product is ready for deployment.

In terms of mapping, the following provides some guidance that may be considered.

- Concept -> Requirements & Architecture, Brainstorm, Plan, Analysis
- Design -> Build, Prototype
- Implementation -> Develop, Construct, Coding
- Integration
- Build -> Compile
- Test -> Validation
- Deployment -> Release, Maintenance

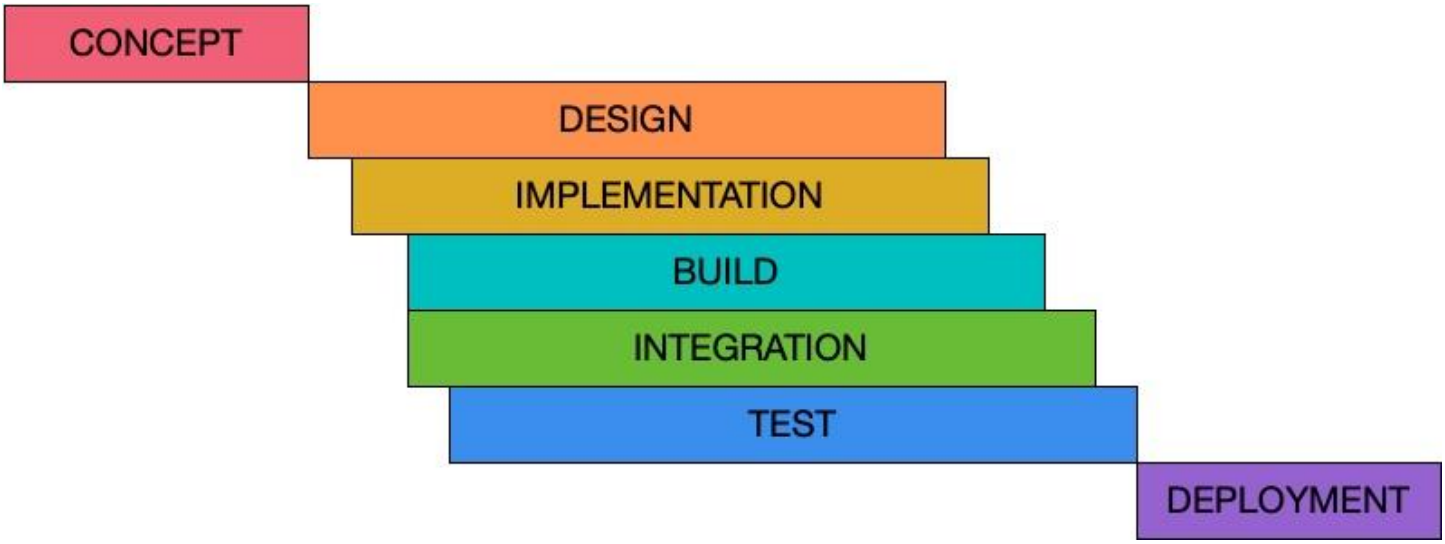


*Figure 1 -Stages of the Software Development Life Cycle*

### 3.1.1 Concept Stage

The Concept Stage is the birthplace for all software development. It is generally started based off an initial idea or need the drives the requirement for a new product or an enhancement to an existing product. This is traditionally where a set of requirements is established to provide guidelines for how the product will address the identified need, as well as creating a conceptual architecture for the product.

Requirements identified during this stage will also include the tools and policies for the development of the software product. This stage includes determinations on things like development tools and languages, repository type with users and roles, and team assignments and structure.

### 3.1.2 Design Stage

The Design Stage builds upon the conceptual architecture created in the Concept Stage and delves into the specific of the architecture. The details of the architecture include not only the software solution itself, but also any and all hardware or software element with which it will communicate. All inputs/outputs to/from the software are identified and the specifics of each are defined.

It is also during this stage that the product is divided into individual modules based upon the defined architecture. The modules are often defined based on integration with a specific piece of hardware, adherence to an industry specification, or a core feature or capability. Similar to the architecture, all inputs/outputs are identified and communications defined.

This stage may be re-entered multiple times during the creation of a software product. It is not uncommon for teams to address a single, high-level module within an architecture and take it to completion before fully defining all other high-level modules. It may also be re-entered as new needs or features are identified over the lifespan of the product or even within its initial development.

### 3.1.3 Implementation Stage

Once the design work has been completed, or at least some portion of it, the Implementation Stage begins. For software products, this typically mean the writing of the code. This stage will also have significant overlap

with the Integration, Build, and Test stages as modular code bases may incrementally progress through all four in a cycle that repeats until the final product is completed.

### 3.1.4 Build Stage

The Build stage is typically iterative and used as periodic checkpoints in the overall development process. With software, the build sequence if often used not only to allow for functional testing of the generated code, but also to identify syntax and logic bugs that exist in the code. Compilers are a first line of defense in helping to discover potential issues through the use of compiler and linker flags used during the build process. The generation of proper and effective build scripts becomes a critical aspect in the creation of binaries with as few potential vulnerabilities as possible.

As with the rest of the development process, the build stage is likely broken up into creation of specific modules, libraries, and executables. The output of individual build steps often becomes the elements that are later integrated into the overall product. A final build step for creation of the end product may re-invoke prior builds or may simply integrate their corresponding output.

### 3.1.5 Integration Stage

The Integration stage includes bringing together both internal and external hardware and software. This stage is likely to be recurring over the life cycle of the product, occurring at specific product milestones or as individual modules are completed or delivered. It is important to note, especially for software, that integration includes any and all code sources not specifically created for and by that product. Even utilizing open-source libraries, such as OpenSSL or Boost, are considered part of the Integration stage by this document.

Integration can also include the combination of software with hardware. Software may be developed, built, and tested on non-native hardware, but at some point, the software must be integrated with the final or native hardware prior to final build, testing, and then deployment.

### 3.1.6 Test Stage

The Test stage is used to verify and validate all aspects of a solution prior to its deployment, in as much as it is mathematically feasible to do so. This includes testing of functional and logical elements of the product, as well as verifying interfaces for proper and sufficient connection and implementation. Presumably testing will also be performed iteratively as each module of the product is completed with a final, exhaustive test(s) conducted once everything all components have been integrated together.

### 3.1.7 Deployment Stage

The Deployment stage manages the release of the initial product, along with all subsequence patches, updates, or revisions of the product over its life. All prior cycles are likely to continue or to be restarted after the beginning of the first Deployment stage, assuming the product is not a one-off solution. As such, the Deployment stage remains until the end-of-life (EOL) for the product.

# 4 Common Life Cycle Adoptions

The SDLC illustrated in Figure 1 provides a very basic and generic instantiation and flow of the defined stages. Over the last two decades, a number of new methodologies have been developed that modify this flow to custom tailor it more for the creation and deployment strategy used by the software provider. Understanding these different approaches is critical to identifying the most significant attacks that might be leveraged against providers.

This section provides a brief overview of some of these new methods. It should be understood that neither this section, nor this document as a whole, is intended to provide any form of analysis of the pros and cons for each method. The information here is provided simply for the purpose of providing a high-level overview of the different approaches used today for software development.

## 4.1 Waterfall

The Waterfall model is conceptually very similar to the one presented previously. However, the major difference is that solutions do not move from one stage to another until all work from the prior stage is completed. This results in implementations that tend to be simpler, but less flexible. Because of the dependencies, this model does not allow for parallelization of efforts between phases. Multiple teams may be involved in a single phase but must be working on the same phase.

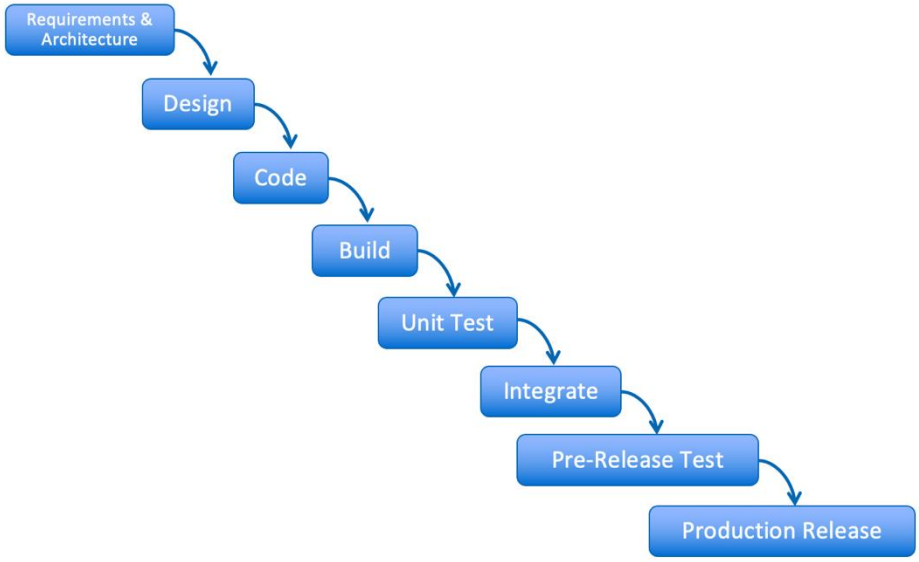


*Figure 2 - Waterfall Development Method*

## 4.2 Iterative

The Iterative model utilizes a repetitive sequence of Design->Code->Build->Test (DCBT) resulting in a final product that is then integrated into its final environment, tested accordingly, and then released to the public. This particular approach provides the ability to modularize a software solution that gradually builds upon each iteration of the DCBT sequence.

Because this approach can be highly modularized, it allows for parallel execution among a number of teams. This often results in a solution made up of pieces from multiple groups. Presumably the teams all follow the same coding and development guidelines or standards and share an overall codebase on a shared repository. However, it often creates multiple targets for attackers and provides a rich set of options for the best place to perform an attack.

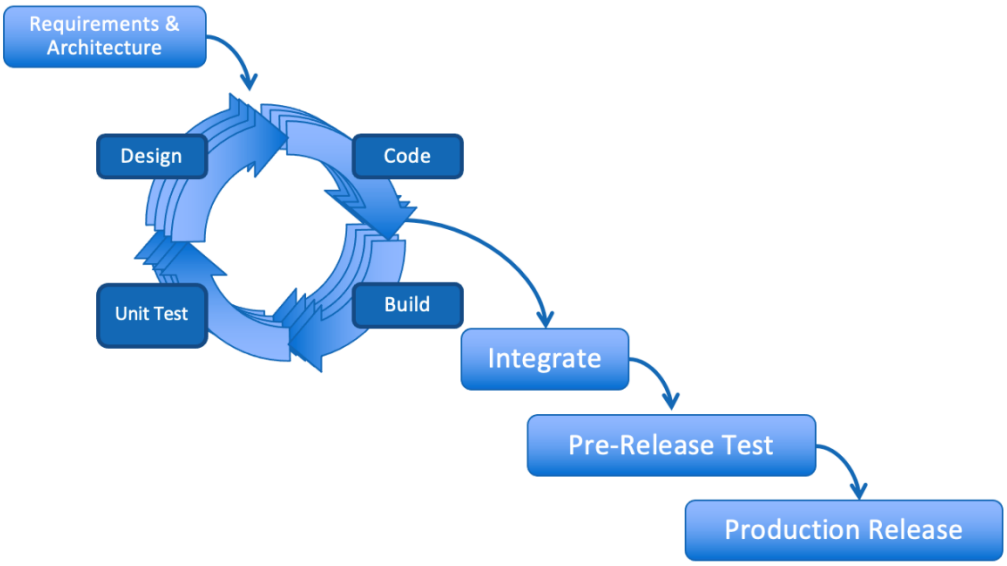


*Figure 3 - Iterative Development Method*

## 4.3 Continuous Integration

This methodology is similar to iterative development, except that here the product is continually evolving with new modules or updates being integrated. With this approach rather than waiting until all modules of the solution are completed to begin integration, it is instead performed each time a module is successfully completed. This results in an intermediate solution that is built upon step by step, but also means an incomplete solution is residing at one or more locations. As a consequence, this approach provides the potential for introducing several additional attack points and expanding the number of potential attackers.

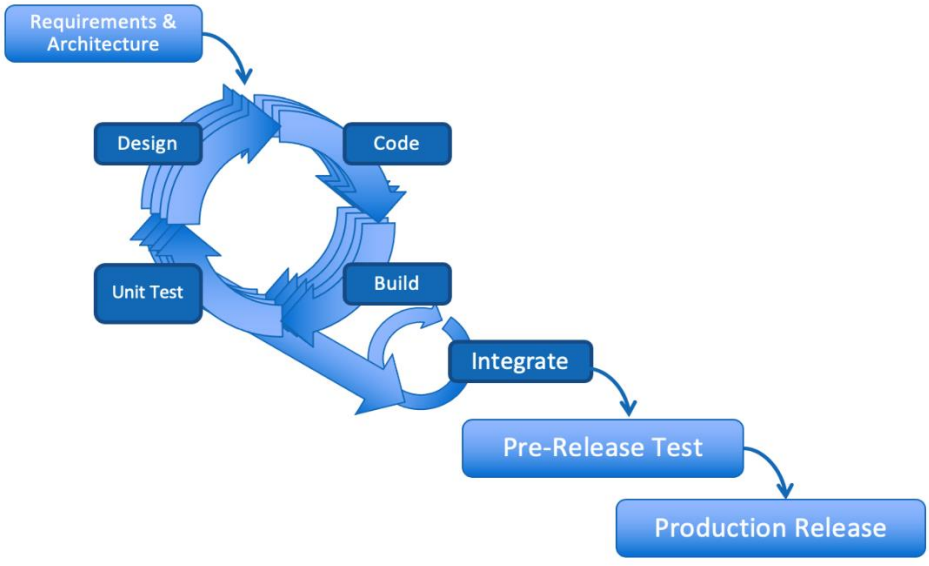


*Figure 4 - Continuous Integration Model*

## 4.4 Continuous Delivery

The next methodology is continuous delivery and is based on the notion of having an end product constantly ready to go and providing delivery as determined by customers. All code needs to be ready to go at all stages. This results in a significantly more automated process as opposed to a more controlled build that might be invoked only based on a pre-defined schedule or with the release of a new update. This may reduce the potential for malicious actions during the build process but puts significantly more dependency upon the security of the build systems themselves. It also results in more build versions as literally every good build is released.

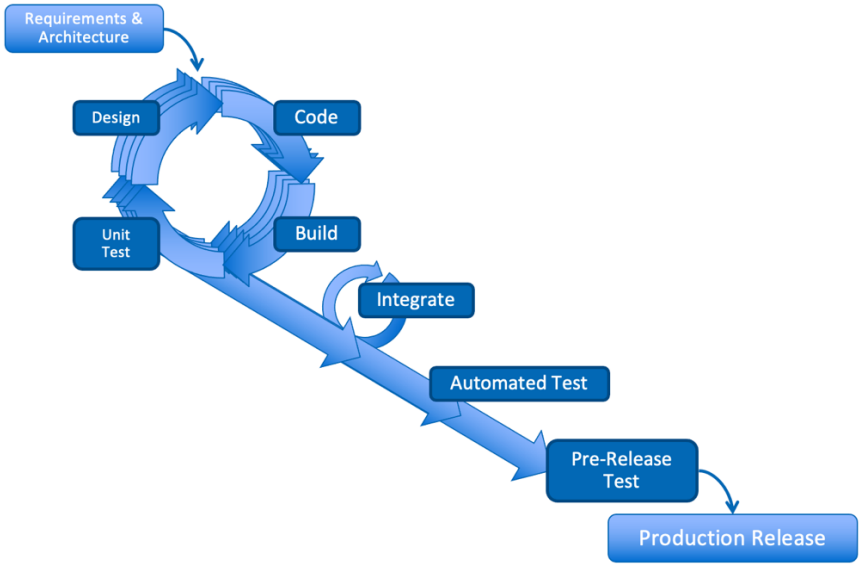


*Figure 5 - Continuous Delivery Model*

## 4.5 Continuous Deployment

The continuous deployment methodology is virtually identical to continuous delivery except that each good build is immediately deployed as opposed to waiting for it to be requested by customers. This has the effect of potentially pushing a vulnerable version to customers automatically, thereby expanding the number of exploitable victims.

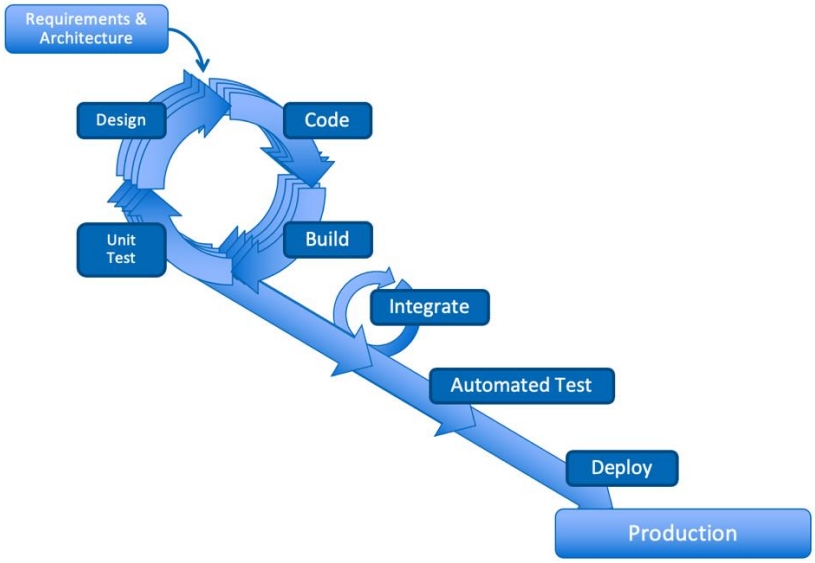


*Figure 6 - Continuous Development Model*

# 5 Threat Identification

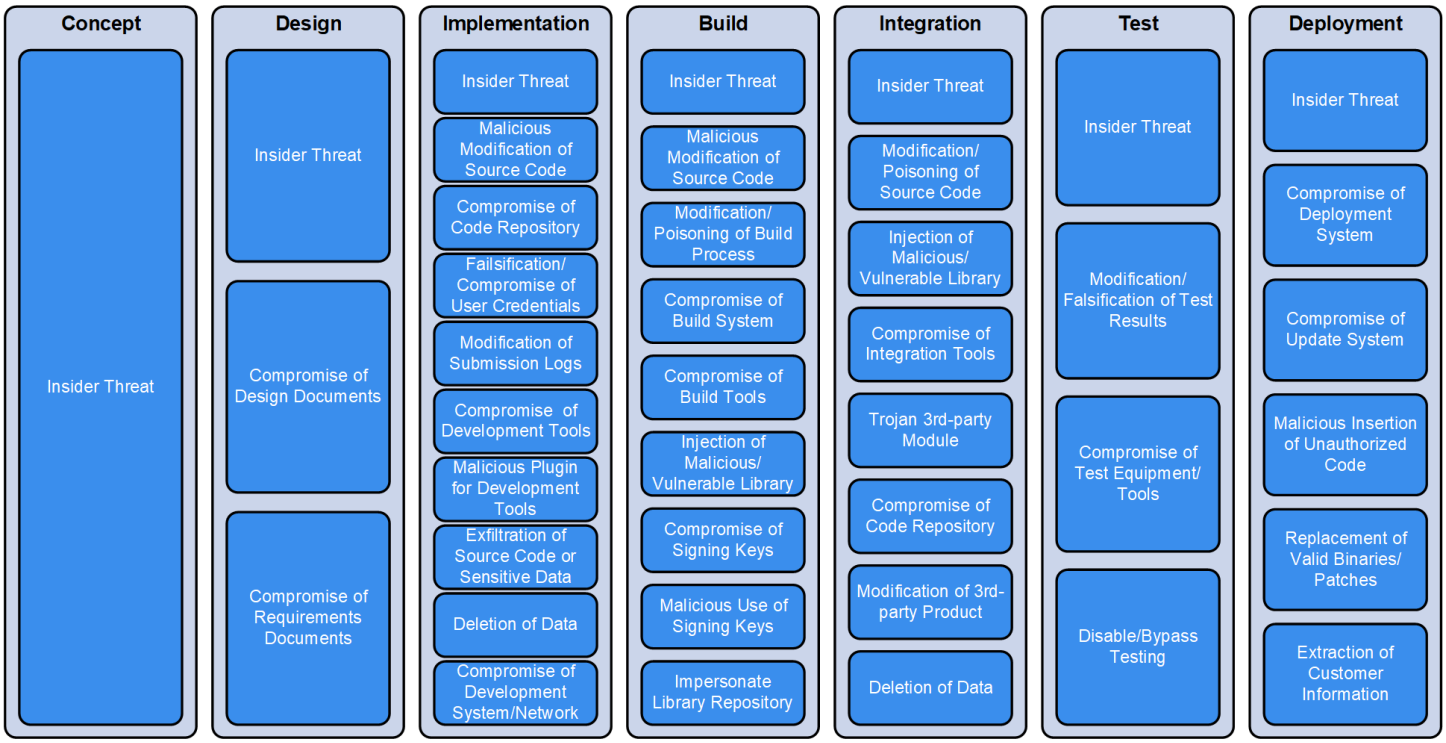


*Figure 7 – Software Development Life Cycle Threat Listing*

## 5.1 Threat Definitions

The remainder of this section is a listing of the definitions of each threat shown in the figure above. These definitions, many of which include examples of attacks, should be used to determine if sufficient mitigations already exist or if a gap has been discovered that will need to be addressed. The threat model presented is based off currently known or suspected attacks and is subject to future additions. As such, it is subject to future additions as new attack vectors are identified.

### 5.1.1 Insider Threat

The threat of a malicious, or even unintentional, insiders is the most pervasive and potential impactful of all threats. It spans all stages of the SDLC and can be realized through virtually any of the attacks identified in this section. Most threats are at least somewhat mitigated via network security solutions, such as intrusion detection and corporate firewalls. Insiders are often able to operate from within a corporate network and thus bypass many of these mitigations.

### 5.1.2 Compromise of Design Documentation

Design document generated during the Design phase represent critical architectural and functional aspect of the end solution. Any malicious modifications to these documents could result in removal or alterations of security features, or introduction of known vulnerabilities or weaknesses into the design. Further, even unauthorized disclosure of these document could provide significant assistance to attackers in discovering attack points not sufficiently protected by any mitigations.

### 5.1.3 Compromise of Requirements Documentation

Requirements documents detail specifics about implementation and integration with a software solution. At the Design phase, many of these requirements are still relatively high-level. Modification of these requirements can result in significant impact to the overall security of the product as individual modules may not maintain cohesive or sufficient security requirements. Modifications also do not have to be malicious and could simply be incidental, but not comprehensive, meaning that the impact of intentional modifications were properly assessed in regard to their impact on the overall product. Further, even unauthorized disclosure of these document could provide significant assistance to attackers in discovering attack points not sufficiently protected by any mitigations.

### 5.1.4 Malicious Modification of Source Code

The source code for software programs is effectively the "crown jewel." The ability of an attacker to modify source code should be of critical concern. If an attacker is able to modify source code, they can introduce vulnerabilities, weaknesses, backdoors, or disfunction, among other things. Depending on where these modifications occur, it may be relatively easy for an attacker to insert the modification without detection.

Modification can potentially happen in a number of different places and across several stages of the SDLC. It could occur on any employee computer hosting the source code, on the repository used to maintain the source code, on build systems that compile the code into libraries or executables, or during transit between any such systems. Such modification may or may not be detected. For instance, if an attacker changes a file on a developer's computer, that developer might not notice the change when they perform their next commit to the repository.

### 5.1.5 Compromise of Code Repository

Code repositories are used to maintain global copies of source code used in development of software products. A compromise of such a repository could result in several possible issues, including modification of source code, leakage of source code, alterations of commit logs, or the addition or removal of authorized users. An attacker could potentially have full control over the repository itself and even use it to attack users as they sync with the repository. The code repository could be local, i.e. on the corporate network, or may be a remote, i.e. external to the corporate network.

### 5.1.6 Falsification/Compromise of User Credentials

User credentials are typically used to identify individuals responsible for modifications to source code, as well as controlling access to share resources. Further, these credentials may also afford certain roles and responsibilities for each user, affording customization of access to specific repositories. Compromise of these credentials could allow an attacker to access repositories, make modifications as a legitimate user, access privileged resources, modify logs, and a variety of other actions. This could also potentially be done if an attacker is able to falsify user credentials, either by duplicating a legitimate credential or creating a fake user credential,

Attackers may also seek to obtain credentials from individuals who have left a company, especially disgruntled former employees. This would potentially allow them to gain access to systems when credentials are not quickly or effectively revoked.

### 5.1.7 Modification of Submission Logs

Submission logs are used to maintain information related to changes in source code and the user responsible for them. This information includes what the modifications were, when they were made, who made them, the

name or IP address of the system used, and comments by the submitted on the nature of the changes. Changes made to any of this information could result in variety of different impacts to the solution, such as deleting information about malicious alterations or inserting false information to implicate an innocent party.

### 5.1.8 Compromise of Development Tools

Design tools, such as Software Development Kits (SDKs) and Integrated Development Environments (IDEs), provide feature-rich interfaces for developer in creation of their software. This includes tools like Microsoft Visual Studio, Apple Xcode, Eclipse, Vim, Emacs, and Netbeans. If an attacker is able to compromise the integrity or configuration of such tools, it would be possible to automatically inject malicious or vulnerable code into products. Detection of such an injection is often difficult as it is not always possible to correlate each line of a resulting library or binary with the file and code that was used to create it.

Additionally, the design tools themselves may be malicious in nature. Installation of the tools often requires administrative privileges and/or the installation of drivers for the operating system. If the tool is malicious, it could use such privilege to install backdoors into the system itself or may simply inject malicious code into whatever product it is used to generate.

### 5.1.9 Malicious Plugin for Development Tools

Many of the design tools mentioned previously also support the ability to add open-source or freely distributed plugins to provide some improvement to the tool's existing capabilities or interface. Such plugins are rarely subjected to extensive security assessments or validated in any form, other than potential integrity checks. Rather than attacking the design tool itself, attackers may leverage a plugin to provide similar capabilities as they are a, relatively speaking, much easier attack vector. This is also true for various other tools, such as application lifecycle management, build systems, code repositories, automation framework system, and testing tools. All such tool typically allow for plugins with varying degrees of security validation.

### 5.1.10 Exfiltration of Source Code or Data

Not all software solutions are open source. As such, exposure of the source code for the solution could result in enhanced security risks as assumptions may occasionally be made that are based on the confidentiality of the code. As a result, if an attacker is able to compromise a developer machine or repository that has the source code, they may be able to extract the code to their own system. This would allow them much greater introspection into how the solution works and allow them to automate some of their vulnerability exploration efforts.

Similar to source code, exfiltration of other data can be critical to a product. This would include information on the tools used to perform a variety of actions, from testing and validation to building the end product. If an attacker is able to garner this information, it can help to refine future attacks or help them identify vulnerable systems. Leakage of configuration information may provide insight into the extensiveness of tests and help in identifying gaps that may be primary attack points.

### 5.1.11 Deletion of Data

Some attackers may simply be looking to disrupt or even bankrupt a company. One of the easiest ways to do that is the destroy the asset itself: the software. If an attacker can gain access to the development network, they may be able to delete repositories or backups that are used to store the source code of the product. This can result in anywhere from minor to catastrophic damage to the company if the attacker is able to delete enough data.

This risk of data loss is also not limited to source code. Similar consequences may occur from the loss of configuration data, provisioning data, cryptographic keys, distributions logs, and customer information.

Virtually any data retained by the company producing the product has some inherent value, the loss of which would bring a variety of consequences.

### 5.1.12 Compromise of Development System/Network

Any system used to perform development work, or to transport associated data, is a potential target for an attacker. Once they are able to gain access to the system, they can use it to then gain access to any data or resources to which that developer or device may be privy. Rather than attack a source code repository or a build system itself, a development system is often substantially easier. This is especially true for mobile development systems.

Additionally, the development network itself may be a target. Several known attacks have occurred due to malicious, 3rd-party devices being connected to unnecessary and/or insufficiently protected, networks. Sure devices are prime targets as they are often installed behind many of the network intrusion solutions designed to keep attackers out of development networks. This is especially true of network equipment, such as routers and intelligent switches.

### 5.1.13 Modification/Poisoning of Build Process

The build process is responsible for taking all source code and libraries and converting them into the final software product. If an attacker is able to modify the build process in any way, they could replace legitimate files with malicious versions or swap a stable library with a known vulnerable one. They might also change compiler flags to make the resulting product weaker in some manner, such as leaving in debug symbols or removing optimizations. This also includes attacks referred to as *poisoning* that are specifically design to alter the tweak the process in order to have it pass testing when it really should not.

Additional threats exist based on the extensiveness of the build process itself, such as whether or not the output is cryptographically encrypted or signed and whether or not that is automated as part of the build process. In such a case, the attacker could leak the keys used, use older or vulnerable keys, or utilize a good key to sign a malicious binary.

### 5.1.14 Compromise of Build System

Compromising the entire build system, as opposed to just the process, could provide a number of additional risks. Rather than just being able to alter the build process, the attacker would have access to all assets accessible by the build system, as well as the potential ability to cover up any nefarious activities. The attacker may be able to bypass user authentication checks, submit bogus signing requests, or generate specially crafted versions of the product.

### 5.1.15 Injection of Malicious/Vulnerable Library

Most software solutions leverage one or more libraries developed by the open-source community or 3rd party vendors. Insertion of a malicious or vulnerable library would result in the vulnerability being incorporated into the end product. These libraries could be uploaded to a globally accessible repository or could be injected using a man-in-the-middle (MITM) attack between a target system and the repository with which it is attempting to communicate.

### 5.1.16 Compromise of Signing Keys

Signing keys are asymmetric private keys that are used to provide a cryptographic verification method to ensure both the integrity of the associated code, as well as to prove the authenticity of the code. If an attacker is able to compromise these keys, they could use the key to sign their own version of the software and pass it

off as a legitimate version for the original software provider. A compromise may occur within a Hardware Security Module (HSM) or standard computer used to generate and store the key.

### 5.1.17 Malicious Use of Signing Keys

As opposed to being compromised, a signing key may also be used in a malicious or unauthorized manner. An attacker may generate a malicious or vulnerable software image and then utilize a legitimate signing key to pass it off as an authentic image from the owner of the key. Such an attack would often not require direct access to the key, but rather only the ability to utilize the key.

### 5.1.18 Impersonate Library Repository

An attacker may seek to impersonate the identity of a legitimate repository. Corruption of domain name service (DNS) entries could afford an attacker the ability to receive communications from devices seeking repository data from their expected source. Such an attack may happen at any point along the communication line between the target system and the repository. Additionally, as mentioned before, an attacker may compromise the repository itself and impersonate its identity.

### 5.1.19 Trojan 3rd-party Module

A trojan module is one that behaves in the expected manner at all times until it receives a specific signal. Such a signal typically takes on the form or a pre-defined sequence of values through some input channel. Once the signal is received, the module could change its operation, output a confidential value, or attempt to lock-up a system. Trojans would most frequently be found in 3rd-party modules, though they are often difficult to detect. Detection would require an extensive analysis of the module in comparison with the original source, which may not be provided, or an exhaustive test of all possible input values and sequence, which is usually mathematically impossible.

### 5.1.20 Modification of 3rd-party Product

Software provided by a 3rd-party may come in a number of different forms via a number of different channels. An attacker may seek to modify the 3rd-party product either in transit from the provider to the vendor or while in storage on the vendor network. If there are no integrity protections of the product, or the integrity is not validated prior to any and all usages, it may be possible for the attacker to replace or alter the product prior to its integration into the final software product.

### 5.1.21 Modification/Falsification of Test Results

Test results are meant to provide evidence of both sufficient and accurate responses to specific input. Modification or falsification of such results by an attacker could allow improper or incomplete products to be released to market that can be later compromised. It may also result in the loss of time, money, or inventory to the product owner.

### 5.1.22 Compromise of Test Equipment/Tools

Compromise of test equipment can lead to a number of potential issues and risks for software products. In addition to the modification or falsification of result, such a compromise could also be used to inject malicious payloads, alter functionality, or otherwise modify software while reporting that is still passes all required tests. Test systems may also have access to critical resources, like code signing keys or source repositories, and thereby be used to perform network attacks against higher value targets from within the network.

### 5.1.23 Disable/Bypass Testing

Disabling or bypass testing procedures could allow insecure or weakened products to be perceived as valid and released to market. Such an ability would not require alteration or modification of test results, but rather make it appear as though such testing had already occurred or was not necessary.

### 5.1.24 Compromise of Deployment System

Deployment systems are used to distribute completed versions of a software product to end customers. A compromise of such a system could lead to a distribution of an unauthorized, modified, or malicious version of the product to customers. It could also provide fairly accurate tracking of customers in order to educate attackers on likely targets and future dates of adoption.

### 5.1.25 Compromise of Update System

Compromise of an update system may have similar consequences and effects as that of a deployment system, though the two systems are not assumed to be the same. Further, a compromised update system may take advantage of presumed trusted communications between the update and target systems to inject malicious code into the application under the guise of being an update.

### 5.1.26 Malicious Insertion of Unauthorized Code

Malicious code may be injecting into existing products prior to distribution. This may be the result of an image that is not signed in its entirety or leveraging a time-of-check, time-of-use (TOCTOU) vulnerability in the validation of the signature. This injection could occur anywhere and at any point after the code is originally build, assuming it is not signed. If it is signed, the injection would take place any point after such signing has occurred.

### 5.1.27 Replacement of Valid Binaries/Patches

Valid binaries or patches for software products may be replaced at a variety of different times, including after deployment of the product itself. Many applications that utilize signing only validate the images during the initial download or installation. After such point, they rely entirely upon user privilege restrictions on the host platform to prevent such replacements. This could result in an attacker replacing valid software with a malicious or vulnerable version that is not verified at any point in the future.

### 5.1.28 Extraction of Customer Information

In many cases, attackers are very interested in knowing exactly what customers are downloading products and which version they are using. This allows them to better coordinate their attacks and to identify potential victims. This type of reconnaissance work is usually performed first in an attempt to determine the value of trying to compromise the supply chain of a given vendor.

# 6 Recommendations for Mitigations

The purpose of this section is to provide information on potential mitigations that can be used to address each of the threats identified in this document. A list of suggested mitigation(s) will be provided, along with three measurements on risk to developer (low, medium, high), likelihood of exploit (low, likely, certain), and impact to customer (minimal, moderate, huge). Companies may enter their own scores for each of the threats based on suggested mitigations and any other mitigations not listed. These scores should help companies to determine their highest risk threats and assist in creation of an investment strategy for enhancing their existing set of mitigations.

### 6.1.1 Insider Threat

| | |
|---|---|
| Suggested mitigations: | • User access controls.<br>• Periodic audit of logs (recommended prior to every release, minimum of monthly).<br>• Multi-party approval for critical operations.<br>• Apply hardening to every host on the environment.<br>• Create layer 2 network segmentation and apply network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanner against the environment.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• Use network monitoring tools (IDS, IPS, etc).<br>• Use DLP (Data Loss Prevention) tools. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.2 Compromise of Design Documentation

| | |
|---|---|
| Suggested mitigations: | • User access controls.<br>• Periodic audit of logs.<br>• Redundant systems and/or backups.<br>• Apply hardening to the hosts (servers and workstation) storing design documents.<br>• Create layer 2 network segmentation between servers storing design documentation and workstations; apply network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanner against servers storing design documentation.<br>• Perform periodic penetration testing against hosts storing design documentation.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA.<br>• Use cryptography (in transit and at rest). |

| | |
|---|---|
| | • Classify properly design documents (Confidential, Top Secret, etc.).<br>• Do proper disposal of physical media by destroying it.<br>• Watermark usage on the documents.<br>• Use DLP (Data Loss Prevention) tools. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.3 Compromise of Requirements Documentation

| | |
|---|---|
| Suggested mitigations: | • User access controls.<br>• Periodic audit of logs.<br>• Redundant systems and/or backups.<br>• Apply hardening to the hosts (servers and workstation) storing requirement documents.<br>• Create layer 2 network segmentation between servers storing requirements documentation and workstations; apply network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanner against servers storing requirements documentation.<br>• Perform periodic penetration testing against hosts storing requirements documentation.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA.<br>• Use cryptography (in transit and at rest).<br>• Classify properly requirement documents (Confidential, Top Secret, etc.).<br>• Do proper disposal of physical media by destroying it.<br>• Watermark usage on the documents.<br>• Use DLP (Data Loss Prevention) tools. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.4 Modification/Poisoning of Source Code

| | |
|---|---|
| Suggested mitigations: | • User access controls.<br>• Periodic audit of logs (recommended prior to every release, minimum of monthly).<br>• Multi-party approval for critical operations.<br>• Apply the principle of least privilege. |

| | |
|---|---|
| | • Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• Apply hardening to the hosts (servers and workstation) storing source code.<br>• Create layer 2 network segmentation between servers storing source code and workstations; apply network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanner against servers storing source code.<br>• Perform periodic penetration testing against hosts storing source code.<br>• Use cryptography (in transit and at rest).<br>• Perform security peer review of every new code, critical patches or big changes before committing to the main branch.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks.<br>• Maintain golden copies of the source code on stand-alone system that can be used for comparison. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.5 Compromise of Code Repository

| | |
|---|---|
| Suggested mitigations: | • Create layer 2 network segmentation between code repositories and workstations; apply network access control (firewall) for ingress and egress network traffic<br>• Restricted network access.<br>• User access controls.<br>• Periodic audit of access and system logs.<br>• Redundant repositories on separate systems (preferably at least one that is stand-alone).<br>• Apply hardening to code repositories.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• Use cryptography (in transit and at rest).<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.6 Falsification/Compromise of User Credentials

| | |
|---|---|
| Suggested mitigations: | • Strict and quick revocation of user credentials after departure or termination.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Periodic audit of access and system logs.<br>• Apply the principle of least privilege.<br>• Use cryptography (in transit and at rest).<br>• Never hardcode credentials into the source code. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.7 Modification of Submission Logs

| | |
|---|---|
| Suggested mitigations: | • User access controls.<br>• Periodic audit of access and system logs.<br>• Redundant systems and/or backups.<br>• Periodic validation of current logs against backups.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Use cryptography (in transit and at rest).<br>• Use a centralized log server.<br>• Allow just append to the log files.<br>• Monitor/perform file integrity checks for the logs. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.8 Compromise of Development Tools

| | |
|---|---|
| Suggested mitigations: | • Only authorize installation of signed and verified development tool packages. |

| | |
|---|---|
| | • Host, and require use of, local network, open-source repositories with development tools.<br>• Create development environment images for use by developers that are validated to ensure tools are not modified.<br>• Apply hardening.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Monitor/perform file integrity checks for configuration and binaries.<br>• Use cryptography (in transit and at rest). |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.9 Malicious Plugin for Development Tools

| | |
|---|---|
| Suggested mitigations: | • Only authorize installation of signed and verified design tool plugins.<br>• Host, and require use of, local network, open-source repositories with design tool plugins.<br>• Create development environment images for use by developers that are validated to ensure tools and plugins are properly authorized.<br>• Apply hardening.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Monitor/perform file integrity checks for configuration and binaries.<br>• Use cryptography (in transit and at rest). |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.10 Exfiltration of Source Code or Data

| | |
|---|---|
| Suggested mitigations: | • Monitor development systems for copy operations over network or removable storage (Use DLP (Data Loss Prevention) tools).<br>• Reduce user access to small amounts of overall source code.<br>• User access controls.<br>• Perform periodic audit of access and system logs.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic penetration testing. |

| | |
|---|---|
| | • Apply the principle of least privilege.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks.<br>• Use cryptography (in transit and at rest). |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.11 Deletion of Data

| | |
|---|---|
| Suggested mitigations: | • Perform periodic (as defined by product) backups of critical data.<br>• Maintain backups on multiple systems, at least one of which is on a separate or stand-alone network or no network at all.<br>• User access controls.<br>• Perform periodic audit of access and system logs.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Apply the principle of least privilege.<br>• Use cryptography (in transit and at rest). |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.12 Modification/Poisoning of Build Process

| | |
|---|---|
| Suggested mitigations: | • Validate each deployment build using a stand-alone system within a 24-hour period from release, preferably before release.<br>• Audit all changes to build scripts/processes on a weekly basis.<br>• Maintain golden copies of build scripts/processes on stand-alone system that can be used for comparison.<br>• Automate non-critical steps (other than things such as handling signing keys, etc), and log them.<br>• Use HSM (hardware security modules) for handling secrets such as production keys and certificates.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege. |

| | |
|---|---|
| | • Use cryptography (in transit and at rest).<br>• Only install and use signed and verified building tools.<br>• Monitor/perform file integrity checks for building configuration and tools.<br>• Perform periodic audit of access and system logs.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• Periodically scan the building system for malware.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.13 Compromise of Build System

| | |
|---|---|
| Suggested mitigations: | • Validate each deployment build using a stand-alone system within a 24-hour period from release, preferably before release.<br>• Create layer 2 network segmentation between building systems and workstations; apply network access control (firewall) for ingress and egress network traffic<br>• Restrict inbound and outbound network traffic on build system to corporate network.<br>• Monitor processes started during each build sequence and audit for variations.<br>• Automate non-critical steps (other than things such as handling signing keys, etc), and log them.<br>• Use HSM (hardware security modules) for handling secrets such as production keys and certificates.<br>• Apply hardening.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Use cryptography (in transit and at rest).<br>• Only install and use signed and verified building tools.<br>• Monitor/perform file integrity checks for building configuration and tools.<br>• Perform periodic audit of access and system logs.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• Periodically scan the building system for malware.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks. |
| Risk to Developer: | |
| Likelihood of exploit: | |

| | |
|---|---|
| Impact to Customer: | |

### 6.1.14 Injection of Malicious/Vulnerable Library

| | |
|---|---|
| Suggested mitigations: | • Maintain verified, authentic libraries on corporate network.<br>• Prohibit use of libraries from sources outside of corporate network.<br>• Use cryptography (in transit and at rest).<br>• Monitor/perform file integrity checks.<br>• Monitor 3rd-party components for vulnerability advisories, EOL (end-of-life) and keep those components fixed/updated.<br>• Scan binaries and release package for malware.<br>• Scan Binaries and 3rd-Party components for known vulnerabilities.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.15 Compromise of Signing Keys

| | |
|---|---|
| Suggested mitigations: | • Utilize separate debug and production keys.<br>• Maintain production keys in hardware security modules (HSM).<br>• Keep any system with a HSM on a stand-alone network.<br>• When possible, use PKI (public key infrastructure) to allow revocation of keys and certificates.<br>• User access controls.<br>• Periodic audit of access and system logs.<br>• Require multi-party approval for all production signing operations.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic penetration testing and red teaming.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.16 Malicious Use of Signing Keys

| | |
|---|---|
| Suggested mitigations: | • Utilize separate debug and production keys.<br>• Maintain production keys in hardware security modules (HSM).<br>• Keep any system with a HSM on a stand-alone network.<br>• Require multi-party approval for all production signing operations.<br>• Apply hardening.<br>• Perform periodic penetration testing and red teaming.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, etc.<br>• When possible, use PKI (public key infrastructure) to allow revocation of keys and certificates. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.17 Impersonate Library Repository

| | |
|---|---|
| Suggested mitigations: | • Maintain repositories and mirrors on corporate network.<br>• Require https connections for all repositories, local or external.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Monitor/perform file and configuration integrity checks.<br>• Implement separated recursive DNS servers for workstations and servers to avoid potential cache poisoning attacks. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.18 Trojan 3rd-party Module

| | |
|---|---|
| Suggested mitigations: | • Require all 3rd-party modules to be signed by the provider.<br>• Periodically scan development environments and 3rd-party components for malware.<br>• Virus scan and verify all modules prior to use in a build process.<br>• Restrict 3rd-party modules to pre-defined collection of trusted providers.<br>• Only use verified 3rd-party modules.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing. |

| | |
|---|---|
| | • Create and keep an updated list of 3rd-party components and its dependencies.<br>• Monitor the 3rd-party components for risks (vulnerability advisories, EOL: end-of-life, malware, etc) and keep those components patched/updated. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.19 Modification of 3rd-party Product

| | |
|---|---|
| Suggested mitigations: | • Require all 3rd-party modules to be signed by the provider.<br>• Periodically scan for malware<br>• Virus scan and verify all modules prior to use in a build process.<br>• Restrict 3rd-party modules to pre-defined collection of trusted providers.<br>• Only use verified 3rd-party modules.<br>• Require HTTPS connections for transmission of all 3rd-party products.<br>• Only install and use signed and verified building tools.<br>• Monitor/perform file integrity checks.<br>• Create and keep an updated list of 3rd-party components and its dependencies.<br>• Monitor the 3rd-party components for risks (vulnerability advisories, EOL: end-of-life, malware, etc) and keep those components patched/updated.<br>• Periodically scan the development environment and 3rd-party components for known vulnerabilities. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.20 Modification/Falsification of Test Results

| | |
|---|---|
| Suggested mitigations: | • Integrity-protect all test results.<br>• Maintain copies of test results on stand-alone system.<br>• Perform periodic audits of test results for discrepancies.<br>• Periodic audit of access and system logs.<br>• User access controls.<br>• Apply the principle of least privilege.<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Use cryptography (in transit and at rest).<br>• Monitor/perform file and configuration integrity checks. |
| Risk to Developer: | |

| | |
|---|---|
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.21 Compromise of Test Equipment/Tools

| | |
|---|---|
| Suggested mitigations: | • Test products on closed network, else create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic<br>• Restrict inbound and outbound network traffic on test equipment to corporate network.<br>• Monitor processes started during each test sequence and audit for variations.<br>• Restrict access to test equipment to minimal personnel.<br>• Apply hardening to the tools and testing environments.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Apply the principle of least privilege.<br>• Implement strong authentication methods such as strong password policy, certificates, 2FA, and physical access control.<br>• Monitor/perform file integrity checks for configuration and binaries. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.22 Disable/Bypass Testing

| | |
|---|---|
| Suggested mitigations: | • Require proof-of-compliance, proof-of-testing, for all products, and versions, prior to release.<br>• Audit testing logs on periodic basis for discrepancies.<br>• Perform regression tests to make sure old/solved vulnerabilities don't appear again. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.23 Compromise of Deployment System

| | |
|---|---|
| Suggested mitigations: | • Utilize redundant deployment systems with different operating system environments.<br>• Perform daily audits of products between different systems.<br>• User access protections with access provided to minimal number of employees (Apply the principle of least privilege).<br>• Apply hardening. |

| | |
|---|---|
| | • Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Use cryptography (in transit and at rest).<br>• Monitor/perform file integrity checks.<br>• Perform periodic audit of access and system logs.<br>• Deploy only signed release packages. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.24 Compromise of Update System

| | |
|---|---|
| Suggested mitigations: | • Utilize redundant update systems with different operating system environments.<br>• Perform daily audits of products between different systems.<br>• User access protections with access provided to minimal number of employees (Apply the principle of least privilege).<br>• Apply hardening.<br>• Create layer 2 network segmentation applying network access control (firewall) for ingress and egress network traffic.<br>• Perform periodic vulnerability scanners.<br>• Perform periodic penetration testing.<br>• Use cryptography (in transit and at rest).<br>• Monitor/perform file integrity checks.<br>• Perform periodic audit of access and system logs.<br>• Deploy only signed updates. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.25 Malicious Insertion of Unauthorized Code

| | |
|---|---|
| Suggested mitigations: | • Sign all software products prior to distribution.<br>• Utilize separate systems for code signing and code distribution.<br>• Audit system logs for attempted access to storage location of software products.<br>• Apply the principle of least privilege.<br>• Monitor/perform file integrity checks.<br>• Scan release packages and updates for malware periodically and before any release. |

| | |
|---|---|
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.26 Replacement of Valid Binaries/Patches

| | |
|---|---|
| Suggested mitigations: | • Sign all software products prior to distribution.<br>• Utilize separate systems for code signing and code distribution.<br>• Audit system logs for attempted access to storage location of software products.<br>• Apply the principle of least privilege.<br>• Monitor/perform file integrity checks.<br>• Redundant systems and/or backups. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

### 6.1.27 Extraction of Customer Information

| | |
|---|---|
| Suggested mitigations: | • Maintain customer information on closed network, else restrict inbound and outbound network traffic to corporate network (if possible, do not store customer information on the deployment and/or update environments).<br>• Audit system logs for attempted access to storage location of software products. |
| Risk to Developer: | |
| Likelihood of exploit: | |
| Impact to Customer: | |

# 7 Conclusion

Threats against our software development process and supply chain continue to expand in both scope and severity. It is imperative that all organizations and product developers carefully and thoroughly assess their existing practices to determine what, if any, security gaps exist and develop strategic plans for appropriate mitigations. The information in this document provides an overview of existing threats and an assessment template to assist organizations and product developers in making decisions on where to invest in improvements.

Companies are encouraged to make this an honest assessment. The effectiveness of this approach is contingent upon an accurate representation of existing mitigations in order to accurately identify gaps. Assuming the existence or sufficiency of mitigations, or denying the existence or relevance of threats, will only lead to greater problems down the road. This is a critical assessment for the continued success of any organization and therefore must be as done as honestly as possible.