Workgroup: Network Working Group
Internet-Draft:
draft-birkholz-scitt-software-use-cases-01
Published: 15 February 2023
Intended Status: Informational
Expires: 19 August 2023
A H. Birkholz Y. Deshpande D. Brooks
uFraunhofer SIT ARM REA
t
h
o
r
s
:
R. Martin B. Knight
MITRE Microsoft

Detailed Software Supply Chain Uses Cases for SCITT

Abstract

This document includes a collection of representative Software Supply Chain Use Case Descriptions. These use cases aim to identify software supply chain problems that the industry faces today and acts as a guideline for developing a comprehensive solution for these classes of scenarios.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-birkholz-scitt-software-use-cases/.

Discussion of this document takes place on the SCITT Working Group mailing list (mailto:scitt@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/scitt/. Subscribe at https://www.ietf.org/mailman/listinfo/scitt/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-scitt/draft-birkholz-scitt-software-supply-chain-use-cases.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 August 2023.

Table of Contents

1. Introduction

Modern software applications are an intricate mix of first-party and third-party code, development practices and tools, deployment methods and infrastructure, and interfaces and protocols. The software supply chain comprises all elements associated with an application's design, development, build, integration, deployment, and maintenance throughout its entire lifecycle. The complexity of software coupled with a lack of lifecycle visibility increases the risks associated with system attack surface and the number of cyber threats capable of harmful impacts, such as exfiltration of data, disruption of operations, and loss of reputation, intellectual property, and financial assets. There is a need for a platform architecture that will allow consumers to know that suppliers maintained appropriate security practices without requiring access to proprietary intellectual property. SCITT-enabled products and analytics solutions will assist in managing compliance and assessing risk to help prevent and detect supply chain attacks across the entire software lifecycle while prioritizing data privacy.

1.1. Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Generic Problem Statement

Supply chain security is a paramount prerequisite to successfully protect consumers and minimize economic, public health, and safety impacts. Supply chain security has historically focused on risk management practices to safeguard logistics, meet compliance regulations, demand forecasts, and optimize inventory. While these elements are foundational to a healthy supply chain, an integrated cyber security-based perspective of the software supply chains remains broadly undefined. Recently, the global community has experienced numerous supply chain attacks targeting weaknesses in software supply chains. As illustrated in Figure 1, a software supply chain attack may leverage one or more lifecycle stages and directly or indirectly target the component.

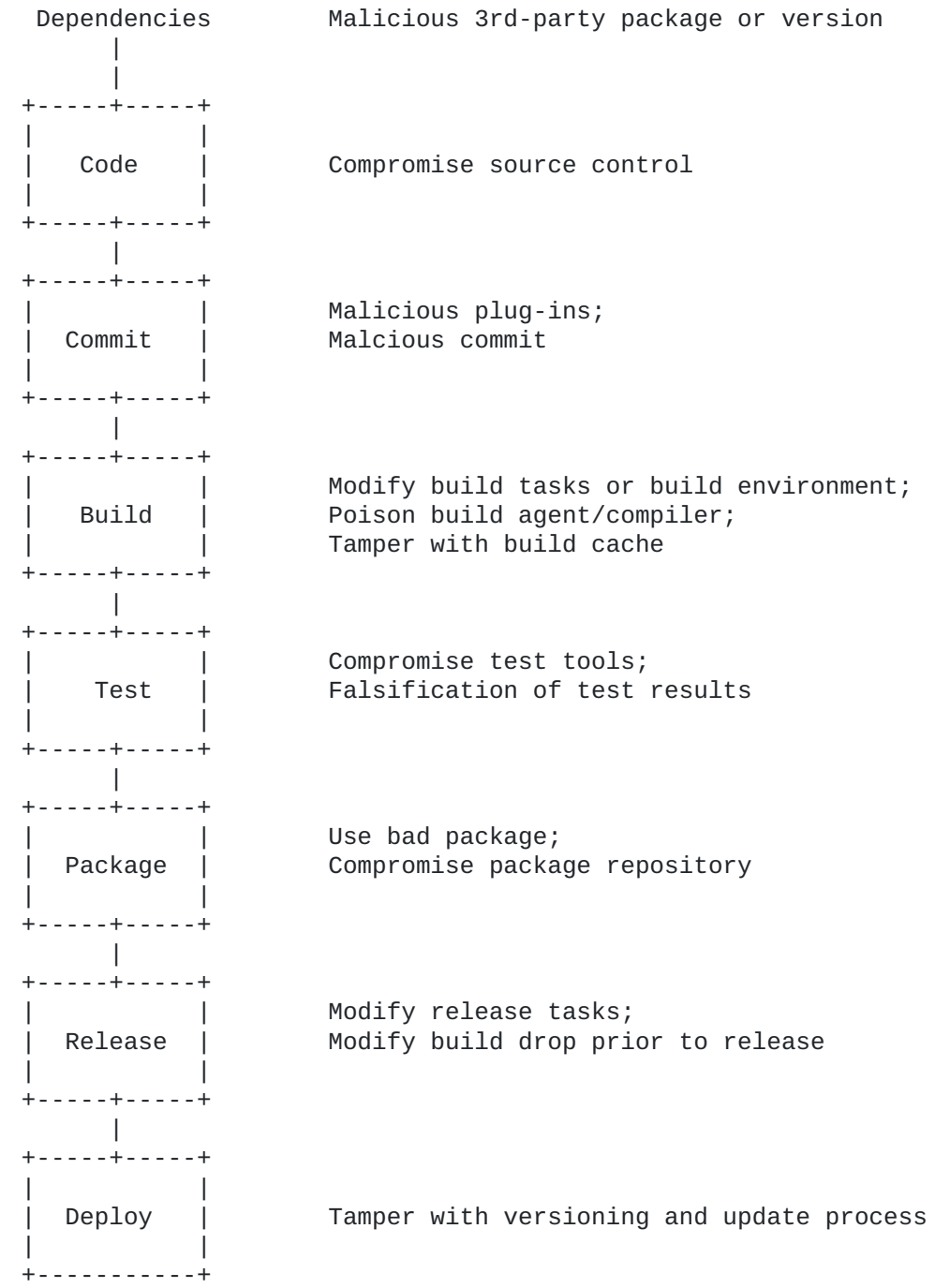

Figure 1: Example Lifecycle Threats

DevSecOps often depends on third-party and open-source solutions. These dependencies can be quite complex throughout the supply chain and render the checking of lifecycle compliance difficult. There is a need for manageable auditability and accountability of digital products. Typically, the range of types of statements about digital products (and their dependencies) is vast, heterogeneous, and can differ between community policy requirements. Taking the type and structure of all statements about digital and products into account might not be possible. Examples of statements may include commit signatures, build environment and parameters, software bill of materials, static and dynamic application security testing results,

fuzz testing results, release approvals, deployment records, vulnerability scan results, and patch logs. In consequence, instead of trying to understand and describe the detailed syntax and semantics of every type of statement about digital products, the SCITT architecture focuses on ensuring statement authenticity, visibility/transparency, and intends to provide scalable accessibility. The following use case illustrates the scope of SCITT and elaborate on the generic problem statement above.

3. Software Supply Chain Use Cases

3.1. Verification that Signing Certificate is Authorized by Supplier

Consumers wish to verify the authenticity and integrity of software they use before installation. To do this today, they rely on the digital signature of the software. This can be misleading, however, as there is no guarantee that the certificate used to sign the software is authorized by the Supplier for signing. For example, a malicious actor may obtain a signing certificate from a reputable organization and use that certificate to sign malicious software. The consumer, believing the software originated from the reputable organization, would then install malicious software.

A consumer of software wants:

* to verify the authenticity and integrity of software they use before installation.

There is no standardized way to:

* enable the consumer to verify that software originated from a 'duly authorized signing party' on behalf of the supplier, and is still valid.

3.2. Multi Stakeholder Evaluation of a Released Software Product

In IT industry it is a common practice that once a software product is released, it is evaluated on various aspects. For example, an auditing company, a code review company or a government body will examine the software product and issue authoritative reports about the product. The end users (consumers or distribution entities) use these report to make an accurate assessment as to whether the software product is deemed fit to use.

There are multiple such authoritative bodies that make such assessments. There is no assurance that all the bodies may be aware of statements from other authoritative entities or actively acknowledge them. Discovery of all sources of such reports and/or identity of the authoritative bodies adds a significant cost to the end user or consumer of the product.

A consumer of released software product wants:

* to offload the burden of identifying all relevant authoritative entities to an entity who does it on their behalf

*to offload the burden to filter from and select all statements that are applicable to a particular release of a multi release software product, to an entity who does this on their behalf

*to make an informed decisions on which authoritative entities to believe based on the best visibility of all authoritative entities possible

There is no standardized way to:

*aggregate large numbers of related statements in one place and discover them

*referencing other statements via a statement

*identifying or discover all (or at least a critical mass) of relevant authoritative entities

3.3. Security Analysis of a Software Product

This use case is a specialization of the use case above.

A released software product is often accompanied by a set of complementary statements about it's security compliance. This gives enough confidence to both producers and consumers that the released software has a good security standard and is suitable to use.

Subsequently, multiple security researchers often run sophisticated security analysis tools on the same product. The intention is to identify any security weaknesses or vulnerabilities in the package.

Initially a particular analysis can identify itself as a simple weakness in a software component. Over a period of time, a statement from another third-party illustrates that the weakness is exposed in the same software component in a way that it is an exploitable vulnerability. The producer of the software product now provides a statement that confirms the linking of software component vulnerability with the software product and also issues an advisory statement on how to mitigate the vulnerability. At first, the producer provides an updated software product that still uses the vulnerable software component but shields the issue in a fashion that inhibits exploitation. Later, A second update of the software product includes a security patch to the affected software component from the software producer. Finally, A third update includes a new release (updated version) of the formerly insecure software component. For this release, both the software product and the affected software component are deemed secure by the producer and consumers.

A consumer of a released software wants:

*to know where to get these security statements from producers and third-parties related to the software product in a timely and unambiguous fashion,

*how to attribute them to an authoritative issuer,

*how to associate the statements in a meaningful manner via a set of well-known semantic relationships, and

*how to consistently, efficiently, and homogeneously check their authenticity.

There is no standardized way to:

*know the various sources of statements,

*how to express the provenance and historicity of statements,

*how to related/link various heterogeneous statements in a simple fashion, and

*check that the statement comes from a source with authority to issue that statement.

3.4. Promotion of a Software Component by multiple entities

A software component source (e.g., a library) released by a certain original producer is becoming popular. The released software component source is accompanied by a statement of authenticity (e.g., a detached signature). Over time, due to its enhanced applicability to various products, there has been an increasing amount of multiple providers of the same software component version on the internet.

Some providers include this particular software component as part of their release package bundle and provide the package with proof of authenticity using their own issuer authority. Some packages include the original statement of authenticity, and some do not. Over time, some providers no longer offer the exact same software component source but pre-compiled software component binaries. Some sources do not provide the exact same software component but include patches and fixes produced by third-parties, as these emerge faster than solutions from the original producer. Due to complex distribution and promotion lifecycle scenarios, the original software component takes myriad forms.

A consumer of a released software wants:

*to understand if a particular provider is actually the original provider or a promoter,

*to know if and how the source, or resulting binary, of a promoted software component differs from the original software component,

*to check the provenance and history of a software component's source back to its origin, and

*to assess whether to trust a promoter or not.

There is no standardized way to:

*to reliably discern a provider that is the original producer from a provider that is a trustworthy promoter or from an illegitimate provider,

* track the provenance path from an original producer to a particular provider

* to check for the trustworthiness of a provider

* to check the integrity of modifications or transformations done by a provider

3.5. Post-Boot Firmware Provenance

In contrast to operating systems or user space software components of a large and complex systems, firmware components are often already executed during boot-cycles before there is an opportunity to authenticate them.

Authentication takes place, for example, by validating a signed artefact against a Reference Integrity Manifest (RIM). Corresponding procedures are often called authenticated, measured, or secure boot. The output of these high assurance boot procedures is often used as input to more complex verification known as remote attestation procedures.

If measurements before execution are not possible, static after-the-fact analysis is required, typically by examining artifacts. When best practices are followed, in such cases measurements (e.g., a hash or digests) are stored in a protected or shielded environment (e.g., TEEs or TPMs). After finishing a boot sequence, these measurements about foundational firmware are retrieved after-the-fact from shielded locations and must be compared to reference values that are part of RIMs. A verifying system appraising the integrity of a boot sequence must identify, locate, retrieve, and authenticate corresponding RIMs.

A consumer of published software wants:

* to easily identify sources for RIMs

* to select appropriate RIMs and download them for the appraisal of measurements

* to be able to assure the authenticity, applicability, and freshness of RIMs over time

There is no standardized way to:

* identify, locate, retrieve and authenticate RIMs in a uniform fashion

* to uniquely identify among multiple potential available RIMs (e.g., by age, source, signing authority, etc.)

* to store RIMs in a fashion that enables their usage in appraisal procedures years after they were created in a secure and believable fashion

3.6. Auditing of Software Product

An organization has established procurement requirements and compliance policies for software use. In order to allow the acquisition and deployment of software in certain security domains of the organization, a check of software quality and characteristics must succeed. Compliance and requirement checking includes audits of the results of organisational procedures and technical procedures, which can originate from checks conducted by the organization itself or checks conducted by trusted third parties. Consecutively, consumers of statements about a released software can be auditors. Examples of procedure results important to audits include: available fresh and applicable code reviews, certification documents (e.g., FIPS or Common Criteria), virus scans, vulnerability disclosure reports (fixed or not fixed), security impact or applicability justification statements. Relevant compliance, requirement, and check result documents originate from various sources and include a wide range of representations and formats.

A consumer of a released software wants:

* to provide methods with different levels of complexity to auditors of a released software
* expects the creator or distributor of released software to enable audit procedures and make corresponding documents visible and available
* the cost of audits to be manageable and scale well
* complete visibility and accessibility to documents required for audits

There is no standardized way to:

* discover and associate relevant documents and check results required for various types of audits
* assert the authenticity and provenance of documents relevant to audits in a deterministic and uniform fashion
* check the validity of identity statements about relevant documents after the fact (when they were made) in a consistent, long-term fashion
* allow for more than one level of complexity of audit procedures (potentially depending on criticality)

3.7. Authentic Software Components in Air-Gapped Infrastructure

Some software is deployed on systems not connected to the Internet. Authenticity checks for off-line systems can occur at time of deployment of released software. Off-line systems require appropriate configuration and maintenance to be able to conduct useful authenticity checks. If the off-line systems in operation are part of constrained node environments, they do not possess the capabilities to process and evaluate all kinds of different authenticity proofs that come with a released software.

A consumer of a released software wants:

- a proof of authenticity that can be checked by an off-line system for vast periods of time after system deployment
- a proof of authenticity to be small and as uniform as possible to allow for application in constrained node environments
- a simple and low cost way to update the configuration of a system component in charge of validity or authenticity checking

There is no standardized way to:

- provide an authenticity proof that can be checked by off-line systems in a simple and uniform fashion
- enable rich systems, regular systems, and constrained systems to conduct authenticity checks via the same procedure / code base
- to verify the authenticity and integrity of software in a fashion that scales from applications such as global open source repositories down to off-line constrained devices

3.8. Firmware Delivery to large set of constrained IoT Devices

Firmware is a critical component for successful execution of any constrained IoT device. It is often the bedrock on which the security story of the devices it powers. For example, personal health monitoring devices (eHealth devices) are generally battery driven and offer health telemetry monitoring, such as temperature, blood pressure, and pulse rate. These devices typically connect to the Internet through an intermediary base station using wireless technologies. Through this connection, the telemetry data and analytics transfer, and devices receive firmware updates when published by the vendor. The public network, open distribution system, and firmware update process create several security challenges.

Consumers and other interested parties of a firmware update ecosystem wants:

- to know that the received firmware for system update is not faulty or malicious
- to know if the signing identity used to assert the authenticity of the firmware is somehow used to sign unintended updates
- to ascertain that the released firmware is not subverted or compromised due to an insider risk - be it malicious or otherwise
- to confirm that the publishers know if their deliverable has been compromised. Can they trust their key protection or audit logging?
- to know how the update client on an instance of a health monitoring system discerns a general update from one specially crafted for just a small subset of a fleet of devices

There is no standardized way to:

- provide an update framework that allows validation of authenticity of firmware revisions
- to verify that the firmware update seen by a single device, is indeed the same as seen by all the devices.
- reliably discern an update that has been signed by the appropriate and intended signing identity
- make an informed judgement on all available information about firmware at the install time. For example, the firmware is still in a good state or otherwise?

3.9. Software Integrator assembling a software product for a smart car

Software Integration is a complex activity. This typically involves getting various software components from multiple suppliers and producing an integrated package deployed as part of device assembly. For example, car manufacturers source integrated software for their autonomous vehicles from third parties that integrates software components from various sources. Integration complexity creates a higher risk of security vulnerabilities to the delivered software.

Consumer of an integrated software wants:

- all components presents in a software product listed, and the ability to identify and retrieve them from a secure and tamper-proof location
- to receive an alert when a vulnerability scan detects a known security issue on a running software component
- verifiable proofs on build process and build environment with all supplier tiers to ensure end to end build quality and security

There is no standardized way to:

- provide a tiered and transparent framework that allows for verification of integrity and authenticity of the integrated software at both component and product level before installation
- notify software integrators of vulnerabilities identified during security scans of running software
- provide valid annotations on build integrity to ensure conformance

4. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://doi.org/10.17487/RFC2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://doi.org/10.17487/RFC8174>.

Authors' Addresses

Henk Birkholz
Fraunhofer Institute for Secure Information Technology
Rheinstrasse 75
64295 Darmstadt
Germany

Email: henk.birkholz@sit.fraunhofer.de

Yogesh Deshpande
ARM

Email: yogesh.deshpande@arm.com

Dick Brooks
REA

Email: dick@reliableenergyanalytics.com

Robert Martin
MITRE

Email: ramartin@mitre.org

Brian Knight
Microsoft

Email: brianknight@microsoft.com