The SolarWinds Attack and Farm-to-table Methods in the Development Process

Mitigating disasters through supply chain security

Felix Pojtinger

2022-01-06

While supply chain attacks have drawn public attention recently due to the high-profile SolarWinds attack, research around the subject is still new due to supply chains of software being fairly simplistic until recently. As modern supply chains are complex sociotechnical systems, fundamental concepts and the current state of supply chain security will be introduced. Special focus is put on the relation of social and technical threat vectors during the modelling of the supply chain, and a technical implementation, in-toto, is introduced as a reference of a supply chain security system. By examining its countermeasures for historical supply chain attacks, in-toto's effectiveness is measured, resulting in protection against almost all historical supply chain attacks.

Contents

1 Introduction

On 13 December 2020, FireEye, a US-based cybersecurity company, detected a large supply chain attack directed at customers of SolarWinds using their Orion monitoring and management solution. The actors of the attack, which are now presumed to be of Russian origin, were able to gain access to data of several high-profile private and public institutions. While the attack was detected and made public by late 2020, it has been active since at least Spring 2020, resulting in widespread lateral movement and data theft. While high-profile attacks against institutions have become more frequent in recent years, the sophistication and threat vector of this attack has shown the need for more advanced modelling techniques and security measures for supply-chain security, which this paper tries to introduce the reader too.

2 Related Work

FireEye provides the initial report on the SolarWinds Attack, detailing the attack's architecture, resulting vulnerabilities and information about the attackers. Sherman and Mark give an overview of the past and presents risks in the supply chain and make general recommendations on means of preventing them. Alsabbagh, Bilal, Kowalski and Stewart present a framework to model a software supply chain from a social and technical perspective, with a focus on highlighting ways to find threats in existing chains. Torres-Arias, Afzali, Kuppusamy, Curtmola and Cappos introduce a concrete implementation of a system with a wholistic approach to supply chain security, as well as an analysis of its level of protection against historical attacks.

3 The SolarWinds Attack

In the case of the SolarWinds attack, the backdoor was built into a digitally signed component of the Orion platform (`SolarWinds.Orion.Core.BusinessLayer.dll`) which communicates with external servers using HTTP. After an initial dormant period of up to two weeks, the malicious component was able to receive commands such as transferring and executing files, profiling a system, disabling services and issuing reboots from command and control infrastructure. Traffic was obfuscated as by masquerading as telemetry communications similar in structure to the Orion Improvement Program. In addition to these obfuscation methods, the malware is sandbox-aware to evade detection by antivirus tools and increase the effort required to do forensics.

In order to deliver the malware to users of SolarWinds Orion, the supply chain was attacked. The attackers were able to compromise SolarWinds' keys and digitally sign a trojanized version of a Windows Installer Patch file, which was distributed using the existing update infrastructure and, after the update process, loaded by the host process. Signature checks did not fail due to the key compromise. Attacks were then able to connect to command and control infrastructure by resolving a subdomain's CNAME record.

4 Overview of Supply Chain Security

While supply chain attacks are not a new occurrence, the SolarWinds attack has forced both developers and security analysts to take the subject more seriously. Historically, most software did not have a particularly complex supply chain; software was limited in size, function and audience, organizations had their own developers and created, for the most part, their own software without many external dependencies. Modern software development has however led to a strong increase in supply chain complexity, as modern software is large enough for it not be manageable by a single organization.

As a result, modern software development is in large part about assembly of existing software, which leads to very long supply chains. A typical "Web 2.0" application could for example include an app server, HTTP server, XML parser, database client, C libraries & compiler, all of which also have their own supply chains. In recent years, in addition to toolchain corruption (such as in the case of XCodeGhost) becoming more and more of an issue, both popular proprietary software and libre software have shown vulnerabilities affecting software released over multiple years (such as for example the "Heartbleed" or "Shellshock" vulnerabilities). In addition to libraries impacting a delivered product, the usage of generated data from tools such as TensorFlow or PyTorch, as well as the use publically available or scraped data such as from Data.gov and Google Open Images leads to new attack vectors on the software supply chain.

Due to multiple high-profile attacks, the interest in means to reduce the risk factors of software supply chains has risen. Simple ways to start reducing risk could, for example, include enforcing rules for suppliers so that they start to follow best practices too, internally ensuring the security and validity of the delivered product by inspection, analysis of the means of delivering the product, and enforcing operational product control so that the product is used securely. From an educational perspective, notarizing the level of security knowledge of employees, ensuring that non-expired education material is used and the credentials of instructors are known, can lead to a reduction in risk. When it comes to checking the suppliers, enforcing the availability or design documents, analysis of attack patterns and the usage of code signing can be used.

At the heart of many supply chain security practices lies the analysis of product delivery methods. Downstreams of software intended for further processing or integration (such as libraries, frameworks or tooling), inherit the consequences of bad upstream security practices. As a result, a consumer must require good security practices by their supplies and asses the risk of delivered products in their used context, which the original authors might not have taken into account (such as the use of software without proper memory protection in environments such as spacecraft, where high radiation levels frequently lead to memory corruption). As a result, the usage of "unchecked" internal supplies should be reduced. This is of particular interest when it comes to integrating external libre software into both own libre software and proprietary software, in which case the recommended way of consuming the external code is to establish a supplier. This can for example be a third party focused on libre software (i.e. RedHat or SUSE), which allows for an evaluation similar to that of proprietary software based on supplier capability, product security, product distribution and operation product control.

5 Modeling the Supply Chain

Fundamentally, software supply chains have a lot in common with physical or hardware supply chains. Supply chains are created by either deploying and using a product directly or by reproducing it as a new product in repetition, and as a result, traditional supply chains tend to have risks such as late product delivery, counterfeits and human errors. Many of these risks also apply to software supply chains and, just like they are in traditional supply chains, must be counteracted using risk management, for which a threat model is a prerequisite. Current practices mostly follow on the classical CIA triad (confidentiality, integrity, availability), but for software supply chains, the security objectives of confidentiality, (data) integrity, source authenticity, availability and non-repudiation are the most important. While organizations such as NIST publish best practices, concrete models of software supply chains are still a rather new subject.

This is in part due to software supply chains requiring the modeling of social and technical behavior in order to model the system and threats to it. Unlike many other security-related models, these domains can't be viewed as purely technical or social, but only as a combination of the two. The basis of modeling the system could for example be ISO 27005, which defines a thread as the potential cause of an incident that might result in harm to systems and organizations. The resulting model should express and capture as many threats as necessary and differentiate between actual threats and potential threats, where the former fulfills the requirements of intention, capability and opportunities, while the latter might lack one or multiple of the three. The detection of potential threats by the model however is important, as they can potentially be prevented by acting early.

Vulnerabilities in the supply chain can be found by analyzing the relations of the integrated elements, which each compose a threat in the system. The model presented by Al Sabbagh and Kowalski, which intents to model both the technical and social elements of supply chain security, introduces two subsystems: A dynamic model of sociotechnical changes (the "sociotechnical" system) and a static model ("security by consensus"). The social subsystem contains culture (collection of values) and structure (distribution of power), while the technical subsystem contains methods and machines, such as technical artifacts. The observation of these four subsystems (culture, structure, methods and machines) determines the security state of the system as a whole; if one part of the system changes, the others must adapt accordingly in order to maintain security (i.e. if new, younger managers arrive, the “structure” parameter might require adjustment). The security by consensus system defines layers of analysis, such as ethical, political and legal, administrative and managerial, operation, application, operating system, and hardware, which makes it possible to determine the correct layers on which to look out for and react to threats.

Both internal or external changes, whether of technical or social nature, will affect security, which requires the systematic rollout of measures. While security frameworks allow analyzing supply chain security using individual layers, only looking at the supply chain as an interconnected sociotechnical system allows reviewers to verify that the layers across the system are secured properly (as issues or protections in one layer might be negated by measures in another layer). A layered approach also allows for the horizontal analysis of supplier’s supply chains and enforcement of measures not at the boundaries of their products and trust, but also based on similarity with one’s own measures on each layer. Comparing the different layers of each supplier’s sociotechnical systems allows for creating an integrated chain of (dis-)trust, where for example the negation of the usefulness of digital signatures for the compiler binaries by compiling unsigned source code or not checking the provided binaries’ signatures is detectable.

When analyzing a supply chain, multiple distinct processes can be found: Supplier sourcing, software development and testing, software packaging and software delivery through network (or, in the case of i.e. embedded software, software product manufacturing and physical software product delivery). During analysis, the products or artifacts which are being sent between the different processes can be the subject of checks. Both supplier sourcing and product delivery naturally link companies and other entities together, which often leads to a generic process like software packaging. In this way, both libre software and proprietary software is being transmitted from supplier sourcing to the software development and testing process, including related secrets and vulnerability information, leading to multiple resulting elements being sent to the user, over the network or physically (such as secrets, vulnerabilities information, source and/or binary package, and the user guide).

In order to find social threats after applying this modelling process, which might exist due to human error or behavioral patterns (intentional or not), the security by consensus model can be used. A supplier can for example deny having sent a product, leading to a non-repudiation issue. Ordered software products also might not arrive in time due to QA problems, leading to an availability issue; also, secrets of outsourced software might be disclosed by employees (such as hard-coded keys or seed values) or the user might make configuration mistakes, such as choosing very short key lengths, if the user guide (which is being distributed as part of the product through the supply chain) has been tampered with.

Technical threats in the supply chain can be analyzed using the bottom three layers of the security by consensus model (hardware, operating system, applications). This could mean vectors such as the software supplier's storage hardware being compromised, allowing externals to inject source code into repositories or packages, outsource software repositories being breached in order to gain access to hard-coded keys (such as API keys in CI/CD environment, configured by i.e. GitLab), or compromising the download site or update system.

Countermeasures to both types of threats can be either social, technical or both. If a non-repudiation issue, such as a recipient of a software product denying receiving it, occurs, a social solution could be to require a third-party notary (on the political and legal layer), while the technical countermeasure might be to use digital signatures to verify the authenticity of the delivered product. To prevent injection of source code during software distribution, a social solution could be to use a third-party escrow, while the technical countermeasure could be to use a VPN or TLS-secured distribution channels.

6 "in-toto", a Framework for Supply Chain Security

in-toto provides a concrete technical implementation of a supply chain security system. It tries to protect against supply chain attacks based on version control systems (such as the breaches of the Linux kernel, Gentoo and Google), build environment (such as the CCleaner breach), software updaters (such as the breaches of Microsoft, Adobe, Google and various Linux distros such as Debian) and others.

Current supply chain security systems are mostly limited to securing individual steps. Technical measures include for example Git commit signing, which allows for controlling what developers can modify in a repo, reproducible builds, which enables (re-)building a package by multiple parties and ensuring a bit-by-bit identical result, and various methods of software delivery, such as APT, DNF or Flatpak. While securing these individual steps is useful and increases security to a certain extent, is leaves open the possibility of modifying the result of a step and passing the modified result to the next step in a chain without the changes being noticed; in other words, there is no way to verify that the correct steps were followed in order or that the artifacts between the individual steps were not tampered with. The problem with the lack of such checks was made apparent by attacks such as the Linux Mint breach, where a compromised web server enabled redirection of installation image download links, resulting in the distro's internal signature checks being negated, even though no key compromise took place. Furthermore, despite fuzzing and static code analysis tools being used more and more during software development, the delivered product rarely includes information about the results of these products for the end-user to verify. The analysis of historical attacks thus shows that solutions designed to secure individual supply chain steps cannot guarantee the security of the supply chain as a whole, and further integration is required to improve it.

in-toto provides such a wholistic approach and tries to enforce the integrity of the entire supply chain.

It requires the declaration and signing of a layout defining how and by whom which steps in the development and release process are to be carried out, thus allowing much greater integration across the supply chain. In addition to this, it also enables involved parties to record actions and create a signed statement for each step in the supply chain ("link metadata"), which makes each step verifiable as in checking if it has been executed appropriately and by the correct party. Doing so is made possible by using checks, which are requirements such as there being no known CVEs in included libraries, or the definition of step intended for translation only being able to create `.po` files in specified directories. In addition to this, in-toto strives to provide security even in the case of a (partial) key compromise by not being a "lose-one, lose-all" solution.

Due to the relative novelty of research into supply chain security, in-toto also provides a significant amount of terminology. It defines a supply chain as being a series of steps performed in order to create and distribute a product, with a step being an operation in the chain taking materials (i.e. source code and artifacts) and creating products (libraries, packages, binaries etc.), which may be executed in parallel or on multiple host for speed or reproducibility. Artifacts in in-toto are defined as materials and products (including both source code and binaries), while byproducts are metadata such as the `STDOUT`, `STDERR` or the return value of a step. Link metadata is a concept central to in-toto; it contains all the materials, products and byproducts for a step and is signed by a functionary, a party performing a step. Functionaries can for example commit code, build software, perform QA or localize docs; they can also be a human entity, providing sign-offs on releases, or a number of hosts providing redundancy or consensus. in-toto also defines the role of the project owner, which is the entity defining the rules for each step in the supply chain and the root of trust. The project owner defines the layout, which is the file defining the steps to be performed by the functionaries, rules for products, byproducts, materials and inspections. The framework also explicitly defines the client, which is the entity that cryptographically validates a delivered product (containing the software, layout and link metadata) by using inspections, which are defined actions to be performed by the client. In real applications, the project owner could be a libre software maintainer, a functionary might be a build farm and a client could be an end user.

As a result, in-toto can protect against supply chain attacks by preventing a number of attack vectors that protecting individual steps alone can not. This includes protection against changing an artifact between two steps (thus preventing modified output from being the input of the next step in a chain), acting as a step without authorization (such as acting as a compiler introducing malware into compiled binaries), providing a delivered product for which steps (such as testing or signing) where not performed, including outdated or vulnerable elements or providing a counterfeit version of the delivered product to users. in-toto managed supply chains can thus guarantee security goals such as supply chain layout integrity (all steps have been performed in the correct order), artifact flow integrity (artifacts can't be changed in between steps), step authentication (steps can only be performed by the intended parties), implementation transparency (existing supply chains need not be changed) and graceful degradation of security properties, so that in the event of key compromise, not all security properties are lost.

Without a key compromise, in-toto protects against a lot of attacks vectors, as even in the case of a breach of infrastructure or communication channels, attackers would not be able to modify artifacts in between two steps or delivered products due to signature validation. Additionally, attackers cannot provide a product with missing or reordered steps due to the embedded layout detecting the tampering, or provide faulty link metadata as the link metadata is signed. As stated earlier, in-toto intends to still retain some level of protection in the case of a key compromise, with the level of protection varying with the severity of the breach. If an individual functionary's key is compromised, an attacker can for example fake that a step has been run (i.e. a signing or testing step) when it actually didn't run, provide a tampered artifact as input to the next step in the chain (product modification), not remove artifacts which should have been removed (i.e. exploitable debug binaries); thus, if a functionary's key is compromised, flow integrity and step authentication are violated (the attacker can fake link metadata), but the attack surface is limited by the rules (permissions) which the functionary has been assigned. Unintended retention would for example only be possible if a `DELETE` rule is missing. This vector can also be somewhat counteracted by requiring multiple parties to do a job, thus requiring a breach of multiple host's keys. In the case of a project owner key compromise, a redefinition of the layout and thus arbitrary supply chain control is possible; due to the infrequent need to access the layout key however it is rarely used, which allows for offline storage, requiring a physical breach and further reducing the attack vector.

In practice, it should be noted that users might respond differently to chain validation errors depending on context. If a user is for example testing out a package in a testing VM, they might choose to ignore the error, while a user deploying software to a production system will probably report the validation failure and not proceed with installations. Actions taken in response to link metadata validation are thus suspected to be similar to those taken in response to package signature validation failures today.

7 Results

When evaluating the effectiveness of supply chain security systems against historical supply chain attacks, it is important to keep the context of their application in mind. Three applications (a Linux distribution, cloud-native deployment and a language-specific package manager) have been analyzed, which each lead to different levels of security.

In the case of Linux distribution, the Debian rebuilders project was chosen by the in-toto maintainers. Debian rebuilders are part of the reproducible builds project, which intents to create bit-by-bit reproducible package definitions, meaning that a source package can be rebuilt on a separate host and result in a binary which is bit-by-bit identical to the original build. In this case, a `apt-transport` for in-toto metadata has been implemented, which is used to provide attestations of the validity of resulting builds using link metadata. This allows for cryptographically asserting that a Debian package has been reproducibly build by k out of n hosts and the build farm; unauthorized modification of a package would require breaching at least k out of n rebuilders.

As for the cloud-native and containerized builds with Jenkins and Kubernetes, it should be noted that such setups require high levels of automation and static configuration; the exporters of in-toto metadata thus must be both host- and infrastructure-agnostic. In the case studied by in-toto, this has lead to the implementation of a Jenkins plugin and a Kubernetes admission controller, which allows for the tracking of all operations in the cluster. Here, pipelines function as coordinators, while workers are functionaries and submit metadata into an in-toto metadata store, with the admission controller validating this information before deploying resources.

The highest level of protection was recorded for the language-specific (Python) package manager; in this case, end to end verification of Python packages was implemented. A tag step outputs Python source code and YAML config files as products, which are then signed using a YubiKey. A builder step receives the signed source code, builds a Python wheel (transferable artifact), and updates metadata. A signer step receives source from the first step and signs it; this step is separate from the builder step due to Python's packaging design allowing potentially dangerous arbitrary code execution. The inspections executed by the client verify whether the build wheel matches the materials of the signer and the source code from the first step, thus providing end to end verification. Furthermore, "The Update Framework" (TUF) is used to provide a higher layer of signed metadata and replay protection and offline keys are used, resulting in breaches of infrastructure not leading to key compromise.

Storage overhead of in-toto stays at manageable levels; in case of the Python application, in-toto accounted for a ~19% increase in repository size. While this level is still manageable, it should be noted that i.e. TUF is able to keep its metadata about 1/4 of this size; the overhead is primarily due to the data being stored as part of the pipeline and large signatures due to safe key lengths adding to the file size, which could be optimized. In terms of network overhead, which is an impact of importance when distributing software updates to many users, in-toto scales linearly with the number of files, not file size. Metadata size amounts to ~44% of package size, but can escalate if many small files are included. Verification layout has been noted to be very low, taking only ~0.6s typically on an i7-6500U-based system with 8 GB of RAM.

8 Summary and Conclusions

Assessment of protection against previous breaches was done in three categories: Control of infrastructure but not functionary keys, control of parts of the infrastructure or keys of a specific functionary, and control of the entire supply chain by compromising the project owner infrastructure, including keys. In tests, the majority of surveyed attacks (23/30) did not include a key compromise. In this case, in-toto's client inspection would have detected the tampering. The Keydnap attack, in which an Apple developer certificate was stolen and used to sign a malicious software package, inspection with in-toto would have detected the attack due to an unauthorized functionary signing the link metadata. In another attack, the developer's SSH key was used to sign a malicious Python package; this would have been prevented with in-toto as the files extracted from the malicious package would not match the source coded recorded in the first step of the chain in case of the Python package manager implementation of in-toto. The CCleaner and RedHat attacks would not have been effective against both the reproducible builds/Debian and Python deployments of in-toto due to multiple hosts building the artifacts, with only the cloud-native deployment lacking a threshold mechanism. In total, both the cloud-native (with 83% prevention as a result of in-toto usage) and reproducible builds/Debian deployments (with 90% prevention) of in-toto would have prevented most historical supply chain attacks. The integration of secure update systems with end-to-end verification as in the Python deployment provides further protection (against 100% of the surveyed historical supply chain attacks in this case), which highlights that further improvement opportunities in supply chain security exist.