in-toto: Practical Software Supply Chain Security

DISSERTATION

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY (Computer Science)

at the

NEW YORK UNIVERSITY

TANDON SCHOOL OF ENGINEERING

by

Santiago Torres-Arias

May 2020

Approved :

Guido Gerig

Department Chair Signature

May 07, 2020

Date

Approved by the Guidance Committee :

Major : Computer Science

[illegible]

Justin Cappos

Associate Professor of
Computer Science and Engineering

May 6, 2020

Reza Curtmola

Reza Curtmola

Professor of
Computer Science

May 6, 2020

[illegible]

Damon McCoy

Assistant Professor of
Computer Science and Engineering

May 6, 2020

[illegible]

Ramesh Karri

Professor of
Electrical and Computer Engineering

May 6, 2020

Vita

Santiago Torres-Arias was born in Mexico City on July 6th. He received his Bachelor of Science in Electrical and Telecommunications Engineering from Universidad Iberoamericana, Mexico. Before joining the Master's of Science at New York University, Tandon School of Engineering, Santiago held positions in the Mexico City Justice Court as a security consultant, and taught High-School level programming in his Alma Matter. On 2013, Santiago joined NYU to pursue his masters and eventually joined the PhD program by Fall 2015.

While on his PhD, Santiago focused his research effort on securing the different stages of software production. This included identifying the common practices of software engineering, as well as development operations and git-operations. This research led him to identifying vulnerabilities and design-flaws in multiple widely-popular software products — such as git, Pacman and tor — and fixing them. Finally, Santiago's work on the now thriving field of *software supply chain security* culminated in the first holistic software supply chain security framework. This academic journey is evidenced in 5 peer-reviewed conference papers (and one under submission), one journal paper and more than two dozen talks in industry, academic and non-for-profit venues alike.

Besides his academic activities relating to software supply chain security, Santiago is a member of the Arch Linux Security team, and has helped publish more than a thousand security advisories. Participates in the reproducible builds community as a developer for their multiple tools. He is also a contributor to many widely used products such as the Linux Kernel, git, the Briar Project, neomutt, and others. Santiago is also a proud member of projects that foster diversity within the communities he belongs, such as Arch Linux Women.

Publications

"Towards Adding Verifiability to Web-Based Git Repositories." H. Afzali, **S. Torres-Arias**, R. Curtmola, J. Cappos. Journal of Computer Security, 2020.

"in-toto: providing farm-to-table security guarantees for bits and bytes." **S. Torres-Arias**, H. Afzali, T. K. Kuppusamy, R. Curtmola, J. Cappos. 28th USENIX Security Symposium (USENIX Security '19) Santa Clara, CA 2019.

"Commit Signatures for Centralized Version Control Systems.", S. Vaidya, **S. Torres-Arias**, R. Curtmola, and J Cappos. 34th ICT Systems Security and Privacy Protection Conference (IFIP SEC '19). Lisbon, Portugal.

"le-git-imate: Towards Verifiable Web-Based Git Repositories." H. Afzali, **S. Torres-Arias**, R. Curtmola, J. Cappos. 13th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS '18). Songbdo, Incheon, Korea.

"On Omitting Commits and Committing Omissions: Preventing Git Metadata Tampering That (Re)introduces Software Vulnerabilities." **S. Torres-Arias**, A. Ammula, R. Curtmola, J. Cappos. 25th USENIX Security Symposium (USENIX Security '16). Austin, TX 2016.

"Diplomat: Using Delegations to Protect Community Repositories." T. Kuppusamy, **S. Torres-Arias**, V. Diaz, J. Cappos. 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16). Santa Clara, CA 2016.

"PolyPasswordHasher: Improving Password Storage Security" **S. Torres-Arias**, J. Cappos. ;login: (the USENIX magazine) pages 18-21, December, 2014.

Acknowledgments

It would be mistaken to believe that a thesis that talks about emergent properties does not show them in ways more than one. Like a secure software supply chain relies on emergent properties of all the nodes involved, I firmly believes that this achievement is not their work alone, but the result of endless hours of help by a countless amount of selfless people. I want to list as many of you as I can in the space I'm provided.

In the topic that relates to Chapter 2, I would like to thank members of the CNCF SIG-Security chapter for their commitment to maintaining and updating the catalog. In particular, Sarah Allen, who initially suggested to broaden the scope of this catalog for the benefit of the community. In addition, I am incredibly thankful to thank Emily Fox and Brandon Lum who, through their thorough review of these compromises and their individual research, have greatly enriched the original efforts of the in-toto team. It was a particularly rewarding experience to walk through time and update my local copy of the catalog to include their contributions in this thesis.

When it comes to the Git security work in Chapter 3, I would like to thank Junio C. Hamano, Jeff King, Eric Sunshine, and the rest of the Git community for their valuable feedback and insight regarding these attacks and their solutions as well as their guidance when exploring Git's internals. For the work in Chapter 4, I would like to thank Holger Levsen, Chris Lamb, and kpcyrd from Reproducible Builds; the Datadog Agent Integrations (especially Ofek Lev) and Product Security teams; as well as Andrew Martin and Luke Bond from Control Plane for their valuable work towards integrating `in-toto` in all these communities. I am also thankful to the in-toto community as a whole as things would not be like they are without their tireless efforts in securing many ecosystems out there. Likewise, I am incredibly thankful to Kay Williams from Microsoft and Kate Stewart from the Linux Foundation for their time discussing ways to deploy in-toto in their ecosystems. I have also found a family in the Arch Linux community, specially in Jelle van der Waa, Levente Polyak and Giancarlo Razzolini, who through their technical insight I was able to keep all of my research grounded in practicality.

Further, and on a more holistic sense, I would like to thank Justin Cappos for his endless trove of knowledge and perspective — every single conversation I have had with him has shifted not only my technical knowledge but, more importantly, my appreciation for all the creative nuances in the academic world. I believe this quality can be extended to every member of my dissertation committee: Reza Curtmola, Damon McCoy and Ramesh Karri.

On an even broader sense — on the in-toto project as a whole — I am also incredibly thankful to Lukas Pühringer and Lois DeLong from the `in-toto` team in particular. I believe that without their unconditional support and their unique insight on the many facets that relate to this thesis. I also think that their passion and professionalism has (ironically) permeated beyond my professional life. Another person in this capacity is Trishank Kuppusamy, who took me under his wing on the early stages of my academic development and even made sure I stayed fed, strong, and sane when I faced seemingly insurmountable hardships.

Zooming out, I believe that I would not be in a position to write this thesis without the help from my longtime friends Daniel Camacho, Ángel Batalla and the whole Foshas, Chinos, and Rojos, who through countless adventures helped me built an invaluable sense of solidarity. That is also the work of Yoshiki Vazquez and David Sevilla, who were tasked with constantly reminding me why I am here. Likewise, I would like to thank Kevin Gallagher, for the back and forth with good, bad, and crazy ideas. Most importantly though, Zahra Ghodsi, who has time and time again picked me up, put me back on my feet, and got me ready for another round.

Finally, as this thesis is also one of the biggest milestones in my life, I cannot help but to look back and see my family. In each of them I have found a value that has guided me through life. In my Abuela Esther lies an admirable strength in spirit. In my aunts Beatriz and Clorinda I have always found a unique intertwining of science, politics and selflessness that has shaped my character and contextualized my knowledge as a force in society. My cousins, starting with Rafael, who will always remind me that witty playfulness is a form of art. Inés, whose fearless drive for answers has always inspired me. I have also always admired my last cousin Mariana's resolve, and I believe I would have never dared to come here in the first place without a success story to cling to. My brother Manuel is probably the single most important figure in my life, and I love to borrow a little bit from his endless source of passion and cosmology — I think it is he who always taught me to look at art as a whole when we walked through the British National Gallery back in 2015. Finally, my mother María Magdalena, the fighter and role model who single-handedly raised me with equal parts love, passion and discipline.

To all of you: may I continue to carry each sliver of knowledge, world view and friendship you gave me, for that makes me whole.

Santiago Torres-Arias

To my mother Maria and my brother Manuel.

ABSTRACT

in-toto: Practical Software Supply Chain Security

by

Santiago Torres-Arias

Advisor: Justin Cappos

Submitted in Partial Fulfillment of the Requirements for

the Degree of Doctor of Philosophy (Computer Science)

May 2020

The software development process, or software supply chain, is quite complex and involves a number of independent actors throughout various organizations and jurisdictions. In most modern supply chains, developers check source code into version control systems, which is in turn compiled into binaries at a build farm, and multiple tests such as dynamic and static analysis, licensing and compliance, security audits, vulnerability scanning among a myriad of other operations are performed. Once all the required actions are carried out, the software is packaged and published for distribution into a delivered product to be consumed by end users.

Unfortunately, software supply chain compromises are common and impactful. An attacker that is able to compromise any single step in the process can maliciously modify the software and harm any of this software's users. According to the Symantec Internet Threat Security Report (ISTR), Software Supply Chain compromise is the fastest growing threat to internet users — which rose 438% from 2017 to 2019. High and low profile companies are affected alike, and the affected includes companies like Docker, NBC news, Microsoft, and RedHat. Protecting against attacks on the software supply chain presents a complicated challenge because, as mentioned above, the ecosystems in which software are made are incredibly varied and a compromise of a simple node in the pipeline often produces a complete subversion of the delivered product.

To tackle this challenge, we took a two-pronged approach: to secure every single operation within this chain (i.e., to build strong links) and build an expressive framework to cryptographically tie ever single step together (i.e., to build a chain out of these links). To do the former, we identified fundamental principles for trustworthy artifact transfer and ensured popular software can provide these principles by fixing vulnerabilities in them. For the latter, we designed in-toto, a framework that cryptographically ensures the integrity of the software supply chain. To do this, in-toto grants the end user the ability to verify the software supply chain from the project's inception to its deployment and enforce compliance of the security policies of each individual step.

This work drives the arc between identifying software supply chain compromises all the way to creating all the principles to prevent these compromises from taking place. The work on securing individual links has crystallized into increasing the security stance of applications such as git, Pacman and the tor browser. In addition, in-toto has been widely used by the time of this publication, as thousands of companies and various open source projects are using in-toto to secure their deployments used by millions of users.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the physical space, the process by which raw materials are sourced, transformed, packaged and distributed as a final product is called a supply chain. These supply chains are a pervasive element of every single industry and, in them lies crucial information about the quality and trustworthiness of said final product.

As consumers of these final products, we are quite familiar with the notion that the practices taken within each step of the chain will have an effect on the product we are about to consume. In fact, transparency in this process is a very natural request from consumers that ask themselves whether to consume a product or not — we intuitively check for expiration dates, tamper-proof seals, and FDA approval indication on medicines, food and drinks. All of these processes are in place both to avoid negligence from actors within the chain but also to avoid supply chain compromises.

Supply chain compromises in the physical world take the face of events such as the Chicago Tylenol Murders [258], where a dozen people died due to cyanide poisoning. This cyanide poisoning was done by the tampering of a yet-unidentified individual who walked into pharmacies, opened Tylenol bottles, introduced the cyanide and walked away. This relatively low-sophistication, yet highly impactful and stealthy attack is the reason now ensure consumables are delivered with a tamper-proof seal. Like tamper-proof seals, security features in the supply chain have evolved with society for as long as supply chains have existed.

However, in the software world, the question remains: how many incidents of the same nature as the Chicago Tylenol Murder have taken place? What measures exist to protect against such? How can we ensure that the existing supply chain practices can extend into the software world?

1.1 Problem Statement

It is no surprise that software follows the a similar process from raw materials into a delivered product. A *software artifact* is sourced from raw materials, such as source code, images, configuration files and environment information and is transformed into a delivered product such as installable media, a container image or a Linux distribution package. These processes are then a *Software Supply Chain* and, like their physical counterparts, it follows that the quality and trustworthiness of a software supply chain rests upon the integrity of the chain. Like tamper-proof seals and lot numbers, the safety of our software products lies in the security practices on each of the individual steps within said chain.

Likewise, events such as the Chicago Tylenol murders have taken place in software: the Symantec Internet Threat Security Report [233, 234] for 2017 and 2018 have included a cumulative increase in software supply chain compromises of more than 400% and affecting companies of varying degrees of scale: software titans such as Microsoft and Google are affected [260, 265] and industries such as financial markets, media, and more have been set back by a software supply chain compromise.

Unfortunately, unlike their physical counterparts, software supply chains, their attacks, and their protection mechanisms have been widely under-explored. As a consequence, there exists a lack of best practices documents, catalogs of compromises and, systems that can be used to protect against them. In such a dire landscape, this thesis is an attempt to tackle these three issues:

1. What are the current software supply chain practices? what effect do these practices affect security? How are these practices exploited to attack software supply chains and their users?
2. What mechanisms, if any, exist to protect against supply chain security compromises?
3. How can we ensure these mechanisms are in place, enforce their verification and protect the integrity of the software supply chain as a whole?

1.2 Contributions

In order to tackle these challenges, we adopted a three-pronged approach. First we needed to understand the nature of these compromises, and thus we analyzed and compiled a catalog of high-profile software supply chain compromises. Second, with the information provided from this catalog, we were able to identify which security properties must each individual step hold in order to protect the integrity of the chain. Third, we created a framework that ensures the integrity of the supply chain as a whole.

1.2.1 Identifying Software Supply Chain Compromises

Cataloging was mostly done through exploratory analysis of compromises that started as early as 2003. We have identified more than 30 different compromises with various degrees of impact — many of them causing millions of dollars in damages [233,234]. These compromises have also affected large companies such as the BBC or Juniper with devastating impacts. In one noteworthy case, attackers were able to decrypt all IPSEC traffic between involved Juniper firewalls for a window of two years [111]. Other cases such as XCodeGhost, allowed attackers to create the first major malware attack on the Apple AppStore by compromising popular applications such as Angry Birds [243]

After identifying these compromises, we needed to understand them in order to prevent further attacks of this nature. To do this, we first explored what was the nature of the compromise within the software supply chain (i.e., which step within the chain was the one affected), and the type of access the attacker had had (e.g., whether the attacker had access to network resources, host machines, credentials or private keys). Once we were able to identify these attacks, we correlated them with existing cataloging techniques, such as MITRE's ATT&CK [198] which let us better understand the shape that these compromises take.

While this work was not published in an academic conference, it has had a wide impact within the industry and the open source communities. This software supply chain compromise catalog is now hosted under the Cloud Native Computing Foundation's sig-security repository, and it boasts more than 66 contributors constantly referring to it and adding new types of compromises.

1.2.2 Protecting Individual Steps

Having identified how attackers exploit software supply chain weaknesses we move onto protecting the software supply chain steps. After creating the aforementioned catalog, we set off to identify the properties that any individual step in the software supply chain must have:

- **Artifact exportability**: A step within the chain must be able to report the artifacts as they are created and consumed. This is important because artifacts created in one step will pass the chain of custody to steps down the line.

- **Artifact authentication**: The ability to authenticate the artifacts created by any party acting on each individual step is necessary so that attackers cannot create artifacts on behalf of other actors.

- **Step-state verifiability**: If a step is stateful (e.g., a version control system), it must provide mechanisms to ensure that its state has not been tampered by attackers.

These principles will ensure that the operations within each individual step on the chain are trustworthy when passing artifacts on to other steps.

With these properties in mind, and after identifying the common targets of compromise, we set off to fix popular tools and steps within the chain. To do so, we have collaborated with various industry and open source initiatives such as Git, Reproducible Builds, PyPI, Datadog, Kubernetes and Jenkins. Chapter 3 will talk about how these principles come to play in the context of git.

1.2.3 Creating the First Whole-Chain Security Framework

Finally, once these principles are provided to a baseline group of projects, we can create the first holistic software supply chain security framework, which provides novel security properties for the software supply chain as a whole. This framework uses the principles provided and allows actors within the chain to create certifications of the actions they performed on the chain. These certifications are designed to generalize to every step in the chain and provide enough semantic information to enforce strong software supply chain integrity and authentication checks.

To enforce these security checks, a trusted party called a project owner is required to create a policy called an in-toto Layout. Within this layout, a series of rules are encoded that include the public keys of the actors that can participate in this chain and where can they participate in. In addition, it also requires this project owner to encode how the artifacts created within each step in the chain can flow. This way, attackers are not able to participate in the chain or inject artifacts as they are passing through the chain of custody.

Chapter 4 will elaborate on the architecture, security analysis and evaluation of the framework as it is used by Datadog. We will also include how all of the projects of the ecosystem are used to provide end-to-end verifiable continuous integration systems, package manager transports to ensure reproducible builds and automated pipeline managers that cryptographically authenticate worker nodes.

The in-toto framework is now hosted under the Cloud Native Computing Framework and it is used to protect millions of companies by thousands of companies worldwide. In addition, `in-toto` is currently facing major improvements to increase its adoption into other types of supply chains, introduced into other standards, such as the Linux Foundation's SPDX 3.0, and tools, such as Google's Grafeas, and even to protect non-software artifacts in different high-assurance scenarios.

Chapter 2

Understanding Software Supply Chains and Their Compromises

Software supply chains, as a concept, are rather new. Before being conceived as such, the software development life-cycle was understood as a series of point operations of disjoint nature. This understanding of the software development life-cycle, has led to countless compromises in the latest decade. This is the consequence of two misconceptions. The first, is that without a whole-picture understanding of the life-cycle, security resources and hardening may be mis-allocated — and remember a chain is only as strong as its weakest link.

The second misconception is that by securing each individual step you will produce a strong chain. The assumption of this emergent property leads to a lack of effort in *linking* steps in the chain together. When this omission occurs, attackers can often provide tampered artifacts to two disjoint systems, regardless of all the security policies in place for the originating artifact.

In understanding compromises as a failure of the supply chain as a whole, lies the key to developing holistic software supply chain security system.

Through this exploration, we will also be able to identify two crucial parts of our bottom up approach to supply chain security. The first, which tools and steps are commonly used in modern software development. The second, which of these are often targeted by attackers and successfully exploited. With this information, we will be able to strategize our supply-chain-hardening efforts.

2.1 Exploring Real-World Software Supply Chains

In order to understand how to protect how software is made, we first must understand how software is actually made. Through this analysis we explored popular software supply chains in search for different insights on what properties do software supply chains have. In particular, we were looking for counter examples to simplistic notions of how software is made as they would be relevant when creating a whole-chain security framework.

Our work focused on surveying popular applications, such as the Django web framework, Live installation media for Linux distributions, Linux packages installed via package managers, and source released from different vendors. We chose these based on their availability, documentation and diversity.

Of these three properties, diversity is an important aspect to underline. The driving principle is that, if we are able to accommodate the most supply chains, it is very likely we will be able to secure the common, less convoluted chains.

As a result, we were able to identify various properties of software supply chains in general that may not appear obvious firsthand. For example, counter to intuition, many software supply chains are non-linear. Further, many supply chains have multiple points of cross-vendor inclusion. We will exemplify some of these counter intuitions next.

2.1.1 Distribution Packaging and Non-Linear Chains

Distribution packaging is a very widespread practice users rely on to set up the tooling on their computers. Linux Operating systems such as Debian, Ubuntu, Arch Linux, and Fedora provide a pre-installed package manager that allows users to install packages curated and maintained by members of the distribution team [5, 59, 78, 79]. In addition, other operating systems such as Microsoft's Windows and Apple's OSX provide Application Stores [93, 197] or developer and automation focused community solutions such as Chocolatey and Brew that mimic the user experience of their Linux counterparts [49, 114, 194].

While the specifics of these tools may change, the broader flow to create a distribution package follows closely the one depicted in Figure2.1. These are:

1. Retrieve *upstream* sources: this is, fetch application code from their source. This is generally a project that is not connected with the operating system, but rather an organization that writes platform-agnostic code [75]. These changes are often fetched from release pages of the specific projects (i.e., hosted on HTTP/ftp servers) or straight from the version control system from the upstream project.

2. Add platform-specific changes (Debianization): Package maintainers will perform changes that are local. These changes are hosted in the form of patches to the upstream releases. This simplifies the flow for updating changes to newer upstream releases. These patches are kept in a distribution-wide version control system such as Git, SVN or Bazaar [25, 90, 107].

3. Prepare and build platform-specific package: this project involves fetching both components (i.e., the upstream release and the changes) and produce a platform installable package such as a .deb or a .pkg file.

Albeit simplistic, this flow highlights the fact that supply chains are not necessarily linear. In fact, most of the packaging practices require to keep the original sources intact. The rationale behind this decision relies mostly on the scrutability of the upstream sources (i.e., differentiate between local and upstream changes) and, as mentioned previously, provide an easy upgrade path of the platform-specific changes made.

Another widespread practice from these types of ecosystems is that of mirroring upstream sources. This is, organizations may keep local copies of the sources fetched from upstream projects. This is important, as it is not uncommon for upstreams to remove releases that are still required by these distributions [211]. This is also the case for private organizations such as Microsoft's Nuget, which also require these sources to be transparently provided to consumers where they to require it [195, 196].

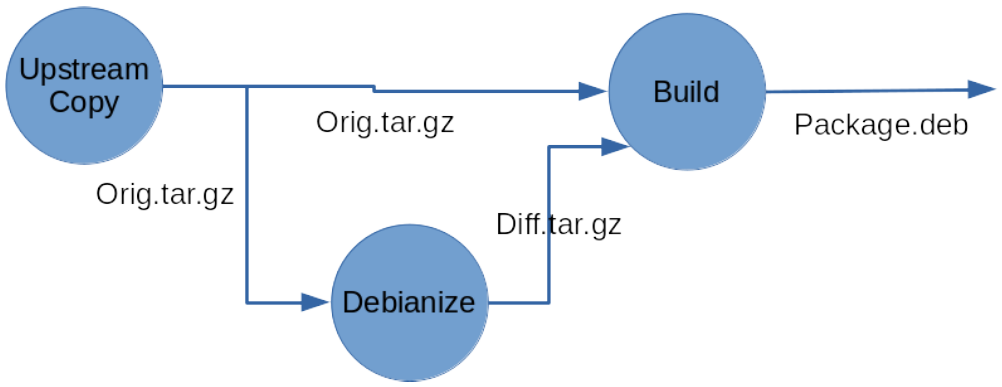


Figure 2.1: The Debian packaging supply chain

Key takeaway Although at first sight it may appear that software supply chains are linear, many supply chains have in fact different artifacts flow in parallel that join in the end.

2.1.2 Live Media and Multiple Fan-In Chains

Live media is a common way to provide users with operating system images for the purpose of testing and installation [77]. In live media, release engineers provide a bootable operating system installation that includes software for common tasks and often include an installation tool [43].

Live media is of particular interest in our exploration because of their high-fan-in cases: oftentimes live-media is comprised of multiple software artifacts from different locations contained in a bootable disk image. Due to this, bootable media tooling fetches artifacts and executes platform-specific configuration scripts on them. Such is the case of common live media tools such as arch-iso [6] the Arch Linux distribution image scripts, and live-build [19] the Debian and Ubuntu distribution tooling.

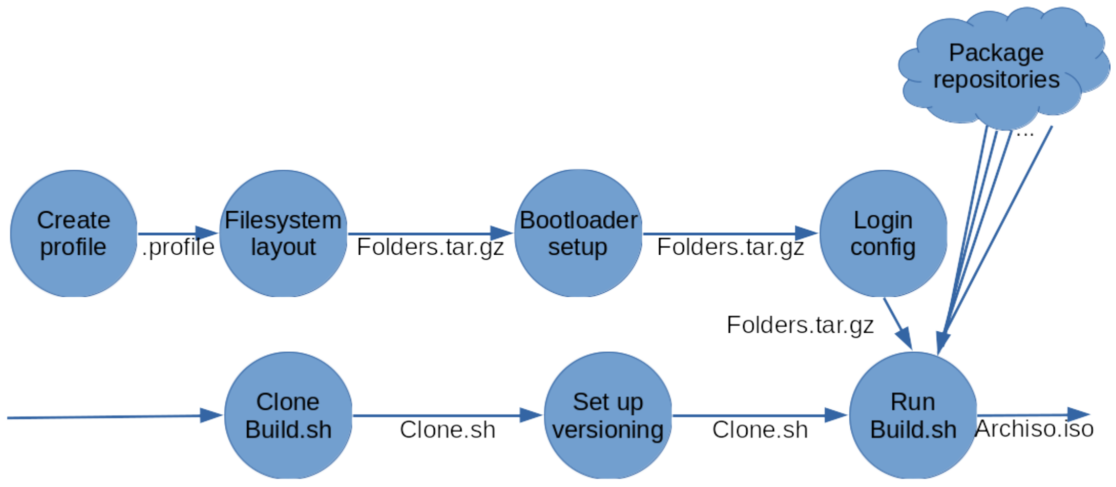


Figure 2.2: The Arch Linux ISO release process

Cross-Organization Bootable Media

Another common use of bootable media are application specific operating disk images. The cases of industry specific toolboxes such Kali Linux [45], a security-oriented operating system or Tails [235] a privacy-focused operating system are also common. These toolboxes are convenient as one-use operating-system deployments. In the case of both, Kali and Tails, the tooling to create them is in fact live-build. The difference between them both will be the per-project configuration files, branding artifacts (e.g., iconography) and any local package modifications.

This is another important use-case of cross-organization supply chains. An upstream vendor is in fact providing the release engineering tooling, and only configuration files are kept by the source artifact providers. This type of deployment is exemplified in Figure2.3

The steps in which these releases are prepared are the following:

1. Pre-release: this includes documenting work such as updating version numbers, or changing release notes for a particular version of the media.

2. Ad-hoc configuration: this is prepare the live-build input files (such as a package list or a list of file-overrides). This is where most of the difference between, for example, Kali and Tails lies.

3. run live-build: run the tool in this context. This will fetch all the packages required for this live-build instance, as well as execute all setup files, custom hooks, and prepare a bootable image.

Key Takeaway Release/deployment (e.g., virtual machines, ISO images or Docker images) media often bundle various software artifacts together. The fan-in of these projects ranges from dozens to hundreds and even thousands of artifacts. The toolchain to create these software artifacts is often its standalone project, whereas the configuration specific deployment media are kept as separate projects.

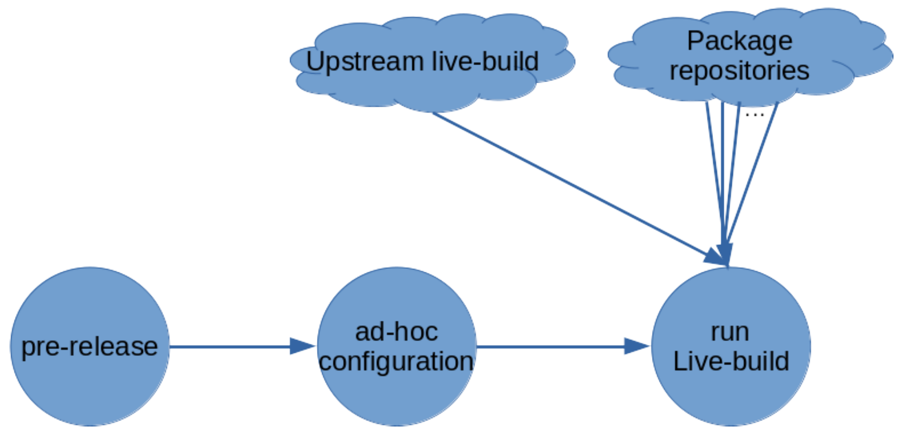


Figure 2.3: The live-build based release (e.g., Tails)

2.1.3 Python Packaging on Long, Linear Chains

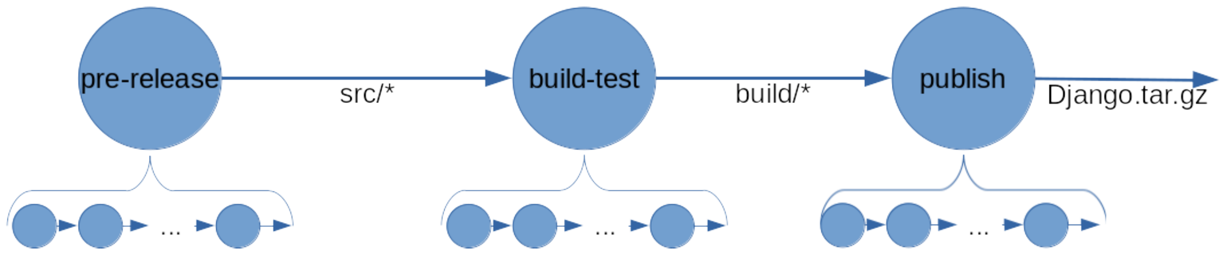


Figure 2.4: The Django PyPI release process

Python packaging is another good example of supply chains that may defeat obvious considerations. In this case, it reveals information about how large organizations may have developed an elaborate process that may require multiple actors and multiple steps, with sub-steps. Such is the case of Django, one of the most popular web frameworks [206], which requires a couple dozen of steps to release their artifacts.

In these cases, there are multiple actors that perform tasks such as pre-releasing (e.g.,, document the release features, change-logs, etc), building and testing, and publishing. These actors may overlap on some activities, but not all actors are allowed to perform all activities.

More interesting, these activities may be separated into sub-teams that will define their own individual practices. For example, a pre-release team may vary their practices depending on the particularities of each release (e.g., an urgent security fix may face less scrutiny on the contents of a change log). These calls are often made by leaders on each sub-team.

This aspect highlights a recursive nature that supply chains may have. In this case, each project is comprised of three individual sub-chains that are linked together to build a final product.

Key Takeaway Highly-linear software supply chains may be subdivided in specific meta-tasks (as seen in the example with Django). This hints that supply chain validation may be recursive, and that each sub-chain may have individual policies that call for transitive trust.

2.1.4 GNU Source Releases With QA Terminal Steps

The case of GNU source releases is a simple, but it is useful to highlight an important aspect of supply chain topologies: those of terminal steps. An intuitive but wrong notion is that the last step on a supply chain is the one providing the delivered product, yet this is not the case when a quality assurance (QA) operation is performed over a release artifact. This case is a common one, but is very evident in source releases such as the GNU project.

The GNU project hosts as much as 396 libre software packages. Most of them provide only source releases, and some use a piece of CI infrastructure (called build-bot) [7, 10], in charge of building and testing these source releases for different processor architectures to ensure the correctness of this source release. For some projects, the built binaries produced by the CI may be available to users, which also highlights the possibility of having two possible products: the source release and an application binary.

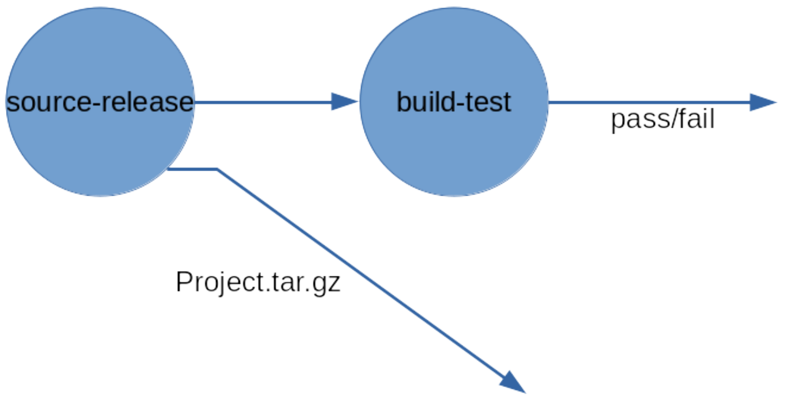


Figure 2.5: The GNU source code release process

Key Takeaway Even on a simple supply chain such as the one depicted on Figure 2.5, the last step in the chain does not produce the artifact users seek. In other words, the *delivered product* of a software supply chain **may not be the last step in the chain**. In addition, a supply chain may have many delivered products.

2.1.5 Supply Chain Tooling Practices

Throughout this exploration, we are able to identify crucial components of the software delivery pipeline. In particular, we are able to notice widespread practices and the common tools to follow them. Of note, we identify the version control system as a central point for software development, and, as we will see in the next section, also a common point of compromise. Of the tools used for version control, we identify git as probably the most widely-deployed solution.

Unfortunately, there is no such mono-culture in the rest of the steps in the chain. From our analysis, we identified more than 15 different build-systems [35] and build toolchains used — sometimes even used in combination. The choice of tooling for build-systems is mostly ad-hoc, and it is generally developed by individual communities and with these communities' needs in mind only. This mono-culture will severely limit the impact of our hardening efforts given their market share.

The same is true for other common steps, such as publishing platforms. In these ecosystems we notice that package managers, programming language package managers, HTTP servers and even version control system hosts such as GitHub provide the ability to provide last-mile delivery. Fortunately, in this case we have identified there are various efforts such as TUF [108], Uptane [182] and Binary Transparency [67] can be used to secure this step in particular.

Finally, there are other steps that are not as common. These include a family of quality assurance tools and testing harnesses. Opposite to the build-system and publishing infrastructure practices, their presence is not as widespread. However, they present a similarly-fragmented tooling market share. These last facts are important, as they *will also underline the need for a tool-agnostic solution to provide holistic security*.

2.2 Analyzing a Corpus of Common Compromises

After understanding how software is built and delivered, we must now explore how these supply chains are often subverted to impact users. This section will focus on exploring a corpus of software supply chain compromises and diving into their specific point of compromise. Through this, we will be able to derive a taxonomy that will also serve to drive the design decisions behind the overarching software supply chain framework.

2.2.1 Types of Compromises

Through work with the Cloud Native Computing Foundation (CNCF) [9] we developed a taxonomy of software supply chain compromises. This taxonomy can be used to understand the common points of failure of software supply chain security and thus are a great starting point for our hardening efforts. We explored 27 different software supply chain compromises and identified 7 different types of compromises. We will elaborate on them next.

Source Code compromise A compromise in a source-code host such as GitHub, an individual repository (such as a self-hosted repository) or a man-in-the-middle connection between a source code repository and a client.

These attacks are prevalent, with 11 instances of them in our survey of 40 attacks (listed on Table 2.1, and with their presence dating as far back as 2003. Although many of these attacks rely on online account compromise (e.g., an attacker obtaining a developer's GitHub credentials), other compromises are more insidious, such as the case of Juniper, in which the infrastructure hosting the source code repository was compromised.

Publishing infrastructure compromise A compromise in publishing infrastructure such as a project's website, GitHub Releases [48], a package repository such as RedHat [64] or a community repository such as PyPI [213].

These compromises are the most common in our survey, accounting to more than 55% of the total number of compromises. While the ecosystem protecting against these types of compromises exist, in many instances they were either not followed at deployment time, or users were tricked into ignoring these mechanisms. This is the case of the Linux Mint compromise [266], where an attacker was able to build a botnet of hundreds of hosts in less than 24 hours.

Negligence and insider threats Attacks caused by insiders such as a rogue employee. This also includes security effects of actors making legitimate mistake or being tricked by an attacker.

These attacks are somewhat uncommon in our survey (only 5% of the attacks), but they have proven to be quite devastating. In many cases, the a proper permission model or a access control mechanism for supply chain artifacts would have mitigated the impact of the compromise significantly. In fact, the existence of roles and the organizational structure of supply chains makes Role-Based Access Control a well suited mechanism to prevent attacks of this nature.

Developer and build toolchain compromises Attacks like these are caused by backdooring compilers (such as Thompson's) as well as developer key/account compromise. These attacks are not entirely common, with only 4 compromises in our survey. However, they may prove to be the most devastating due to the difficulties that arise when auditing the resulting artifacts on these toolchains.

As we will see in Chapter4, mechanisms as reproducible builds [65] and thresholding mechanisms on these operations may prove useful to minimizing the impact of these compromises.

Multi-step compromise these are more devastating attacks in which an attacker can perform a combination of the compromises outlined above. These are rather uncommon (with only one of them in our survey), but their existence hints that there needs to be an overarching solution to protect against these compromises.

Type of Compromise	Count	Percentage
Source Code	11	27.5%
Publishing Infrastructure	22	55%
Negligence/Insider	2	5%
Developer and Toolchain	4	10%
Multiple Step	1	2.5%
Total	40	100%

Table 2.1: Breakdown of compromise types

2.3 Conclusion

This section lays the groundwork to understand software supply chain security compromises in the wild. Through this lens, we are able to make principled decisions on how supply chains look, what are the commonly-used toolchains and techniques to produce software today. Likewise, the catalog of supply chain compromises allows us to understand the security limitations of the current deployment techniques, which will enable us to create a system that can protect them effectively.

Chapter 3

Protecting Individual Links: a Case Study on Git

3.1 Introduction

A Version Control System (VCS) is a crucial component of any large software development project, presenting to developers fundamental features that aid in the improvement and maintenance of a project's codebase. These features include allowing multiple developers to collaboratively create and modify software, the ability to roll back to previous versions of the project if needed, and a documentation of all actions, thus tying changes in files to their authors. In this manner, the VCS maintains a progressive history of a project and helps ensure the integrity of the software.

Unfortunately, attackers often break into projects' VCSs and modify the source code to compromise hosts who install this software. When this happens, an attacker can introduce vulnerable changes by adding (e.g., adding a backdoor), or removing certain elements from a project's history (e.g., a security patch) if he or she acquires write access to the repository. By doing this, attackers are usually able to compromise a large number of hosts at once [46, 101, 115, 143, 168, 187, 260, 263, 264]. For example, the Free Software Foundation's repository was controlled by hackers for more than two months, serving potentially backdoored versions of GNU software to millions of users [150].

The existing security measures on VCSs, such as commit signing and push certificates [26, 154], provide limited protection. While these mechanisms prevent an attacker from tampering with the contents of a file, they do not prevent an attacker from modifying the repository's metadata. Hence, these defenses fail to protect against many impactful attacks.

In this work, we reveal several new types of attacks against Git, a popular VCS. We collectively call these attacks *metadata manipulation attacks* in which Git metadata is modified to provide inconsistent and incorrect views of the state of a repository to developers. These attacks can be thought of as *reconcilable fork attacks* because the attacker can cause a developer's version of the repository to be inconsistent just for a finite window of time — only long enough to trick a developer into committing the wrong action — and leave no trace of the attack behind.

The impact of an attack of this nature can be substantial. By modifying the right metadata, an attacker can remove security patches, merge experimental code into a production branch, withhold changes from certain users before a release, or trick users and tools into installing a different version than the one requested to the VCS. To make matters worse, the attacker only requires a few resources to achieve his or her malicious goals.

We have submitted a vulnerability disclosure to CERT and the GitHub security team describing the following scenario: an attacker capable of performing a man-in-the-middle attack between a GitHub [27] server and a developer using pip to install Django (a popular website framework) can trick the developer into installing a vulnerable version simply by replacing one metadata file with another. Even though Git verifies that the signature in Git objects is correct, it has no mechanism to ensure it has retrieved the correct object. This type of attack enables a malicious party to strike any system that can retrieve packages from Git repositories for installation, including Node's NPM [51], Python's pip [212], Apache Maven [3], Rust's cargo [8], and OCaml's OPAM [53]. As such, it could potentially affecting hundreds of thousands of client devices.

To mitigate metadata manipulation attacks, we designed and implemented a client-only, backwards-compatible solution that introduces only minimal overhead. By storing signed reference state and developer information on the server, multiple developers are able to verify and share the state of the repository at all times. When our mechanism is in place, Git metadata manipulation attacks are detected. We have presented these issues to the Git developer community and prepared patches — some of which are already integrated into Git — to fix them in upcoming versions of Git.

In summary, we make the following contributions:

- We identify and describe metadata manipulation attacks, a new class of attacks against Git. We show these attacks can have a significant practical impact on Git repositories.
- We design a defense scheme to combat metadata manipulation attacks by having Git developers share their perception of the repository state with their peers through a signed log that captures their history of operations.
- We implement the defense scheme and study its efficiency. An evaluation shows that it incurs a smaller storage overhead than push certificates, one of Git's security mechanisms. If our solution is integrated in Git, the network communication and end-to-end delay overhead should be negligible. Our solution does not require server side software changes

and can be used today with existing Git hosting solutions, such as GitHub, GitLab, or Bitbucket.

3.2 Background

3.2.1 Overview of Git

In order to understand how Git metadata manipulation attacks take place, we must first define Git-specific terminology, as well as some usage models of the tool itself.

Git is a distributed VCS that aids in the development of software projects by giving each user a local copy of the relevant development history, and by propagating changes made by developers (or their history) between such repositories. Essential to the version history of code committed to a Git repository are `commit` objects, which contain metadata about who committed the code, when it was committed, pointers to the previous commit object, (the `parent` commit) and pointers to the objects (e.g., a file) that contain the actual committed code.

Branches serve as "pointers" to specific commit objects, and to the development history that preceded each commit. They are often used to provide conceptual separation of different histories. For example, a branch titled "update-hash-method" will only contain objects that modify the hash method used in a project. When a developer adds a new commit to the commit chain pointed to by a branch, the branch is moved forward.

Inside Git, branches are implemented using "reference" files, that only contain the SHA1 hash of a target commit. The same format is used for Git tags, which are meant to point to a static point in the project's history. Both tags and branches live in the `.git/refs` folder.

Git users `commit` changes to their local repositories, and employ three main commands to propagate changes between repositories: `fetch`, to retrieve commits by other developers from a remote repository; `merge`, to merge two change-sets into a single history; and `push`, to send local commits from a local repository to a remote repository. Other common commands may consist of two or more of these commands performed in conjunction (e.g., *pull* is both a `fetch` and a `merge`). Consider the following example:

Alice is working on a popular software project and is using Git to track and develop her application. Alice will probably host a "blessed" copy of her repository in one provider (e.g., GitHub or GitLab) for everyone to clone, and from which the application will eventually be built. In her computer, she will keep a *clone* (or copy) of the remote repository to work on a new feature. To work on this feature, she will create a new **branch**, `handle-unicode-files` that will diverge from the `master` branch from now on. As she modifies files and updates the codebase, she commits – locally – and the updates will be added to the new branch in her local clone. Once Alice is done adding the feature, she will *push* her local commits to the remote server and request a colleague to review and *merge* her changes into the master branch. When the changes are merged, Alice's commits will become part of the master history and, on the next release cycle, they will be shipped in the new version of the software.

Git security features

To ensure the integrity of the repository's history, Git incorporates several security features that provide a basic defense layer:

- Each commit object contains a cryptographic hash of its parent commit. In addition, the name of the file that contains the commit object is the cryptographic hash of the file's contents. This creates a hash chain between commits and ensures that the history of commits cannot be altered arbitrarily without being detected.
- Users have the option to cryptographically sign a commit (a digital signature is added to the commit object) using a GPG key. This allows an auditor to unequivocally identify the user who committed code and prevents users from repudiating their commits.
- A signed certificate of the references can be pushed to a remote repository. This "push certificate" solution addresses man-in-the-middle attacks where the user and a well-behaving server can vouch for the existence of a push operation.

3.3 Threat Model and Security Guarantees

We make the following assumptions about the threat model our scheme is designed to protect against:

- Developers use the existing Git signing mechanisms whenever performing an operation in Git to stop an attacker from tampering with files.
- An attacker cannot compromise a developer's key or get other developers to accept that a key controlled by an attacker belongs to a legitimate developer. Alternatively, should an attacker control such a key (e.g., an insider attack), he or she may not want to have an attack attributed to him- or herself and would thus be unwilling to sign data they have tampered with using their key.
- The attacker can read and modify any files on the repository, either directly (*i.e.* a compromised repository or a malicious developer) or indirectly (*i.e.*, through MITM attacks and using Git's interface to trick honest users into doing it).
- The attacker does not want to alert developers that an attack has occurred. This may lead to out-of-band mechanisms to validate the attacked repository [219].

This threat model covers a few common attack scenarios. First of all, an attacker could have compromised a software repository, an unfortunately common occurrence [46, 101, 115, 143, 150, 168, 187, 260, 263, 264]. Even if the repository is not compromised, an attacker could act as a man-in-the-middle by intercepting traffic destined for the repository (e.g., by forging SSL certificates [96, 97, 103, 169, 224, 245, 249, 259, 262]). However, an attacker is not limited to these strategies. As we will show later, a malicious developer can perform many of the same attacks *without using their signing key*. This means that it is feasible for a developer inside an organization to launch these attacks and not be detected.

Note that in all cases, the developers have known signing keys to commit, push, and verify information.

3.3.1 Security Guarantees

Answering to this threat model, the goal of a successful defensive system should be to enforce the following:

- **Prevent modification of committed data**: If a file is committed, an attacker should not be able to modify the file's contents without being detected.

- **Ensure consistent repository state**: All developers using a repository should see the same state. The repository should not be able to equivocate and provide different commits to different developers.

- **Ensure repository state freshness:** The repository should provide the latest commits to each developer.

As we will show later, Git's existing security mechanisms fail to handle the last two properties. The existing signing mechanism for Git does enable developers to detect modification of committed data, because the changed data will not be correctly signed. However, due to weaknesses in handling the other properties, an attacker can omit security patches, merge experimental features into production, or serve versions of software with known vulnerabilities.

An attacker is successful if he or she is able to break any of these properties without being detected by the developers. So, an attacker who controls the repository could block a developer from pushing an update by pretending the repository is offline. However, since the developer receives an error, it is obvious that an attack is occurring and therefore is easy to detect. Similarly, this also precludes *irreconcilable fork attacks* where two sets of developers must be permanently segregated from that point forward. Since developers typically communicate through multiple channels, such as issue trackers, email, and task management software, it will quickly become apparent that fixes are not being merged into the master branch. (Most projects have a tightly integrated team, usually a single person, who integrates changes into the master branch, which further ensures this attack will be caught.) For these reasons, we do not focus on attacks that involve a trivial denial of service or an irreconcilable fork because they are easy to detect in practice.

3.4 Metadata Manipulation Attacks

Even when developers use Git commit signing, there is still a substantial attack surface. We have identified a new class of attacks that involve manipulation of Git metadata stored in the `.git/refs` directory of each repository. We emphasize that, unlike Git commits that can be cryptographically signed, there are no mechanisms in Git to protect this metadata. As such, the metadata can be tampered with to cause developers to perceive different states of the repository, which can coerce or trick them into performing unintended operations in the repository. We also note that a solution that simply requires users to sign Git metadata has serious limitations (as described in Sec. 3.5.2).

Unlike many systems where equivocation is likely to be noticed immediately by participants, Git's use of branches hides different views of the repository from developers. In many development environments, developers only have copies of branches that they are working on stored locally on their system, which makes it easy for a malicious repository to equivocate and show different views to different developers.

In Git, a branch is represented by a file that contains the SHA1 checksum of a commit object (under benign circumstances, this object is the latest commit on that branch). We will refer to such files as *branch references*. All the branch references are stored in the directory `.git/refs/heads/`, with the name of the branch as the filename. For example, a branch "hotfix" is represented by the file `.git/refs/heads/hotfix`.

We discovered that it is straightforward for an attacker to manipulate information about branches by simply changing contents in a reference file to point to any other commit object. *Modifying the branch reference can be easily performed with a text editor and requires no sophistication.* Specifically, we show three approaches to achieve this, all of them being captured by our adversarial model. *First*, an attacker who has compromised a Git repository and has write access to it, can directly modify the metadata files. *Second*, an attacker can perform an MiTM attack by temporarily redirecting a victim's traffic to a fake repository serving tampered metadata, and then reestablishing traffic so the victim propagates the vulnerable changes to the genuine repository (in Appendix A, we describe a proof-of-concept attack against GitHub based on this approach). *Third*, a malicious developer can take advantage of the fact that Git metadata is synchronized between local and remote repositories. The developer manipulates the Git metadata in her local repository, which is then propagated to the (main) remote repository.

It is also possible to extend these attacks for Git tags. Although a Git tag is technically a

Git tag object that can be signed the same way as a commit object, an attacker can target the *reference* pointing to a tag. Tag references are stored in the directory `.git/refs/tags/` and work similarly to branch references, in that they are primarily a file containing the SHA1 of a Git tag object that points to a Git commit object. Although Git tags are conceptually different — they only represent a fixed point (e.g., a major release version) in the projects history — they can be exploited in the same way, because Git has no mechanism to protect either branch or tag references.

We have validated the attacks against a standard Git server and also the GitHub, GitLab and other popular Git hosting services.

Based on their effect on the state of the repository, we identify three types of metadata manipulation attacks:

- **Teleport Attacks:** These attacks modify a Git reference so that it points to an arbitrary object, different from the one originally intended. The reference can be a branch reference or a tag reference.
- **Rollback Attacks:** These attacks modify a Git branch reference so that it points to an older commit object from the same branch, thus providing clients with a view in which one or more of the latest branch commits are missing.
- **Deletion Attacks:** These attacks remove branch or tag references, which in turns leads to the complete removal of an entire branch, or removal of an entire release referred to by a tag.

We use the following setup to present the details of these attacks. A Git server is hosting the main repository and several developers who have their own local repositories have permission to fetch/push from/to any branch of the main repository, including the master branch. For commit objects, we use a naming convention that captures the temporal ordering of the commits. For example, if a repository has commits C0, C1, C2, this means that they were committed in the order C0, C1, C2.

3.4.1 Teleport Attacks

We identified two teleport attacks: *branch teleport* and *tag teleport* attacks.

Branch Teleport Attacks. These attacks modify the branch reference so that it points to an arbitrary commit object on a different branch. Although we illustrate the attacks for the master branch, they are applicable to any branch, since none of the branch reference metadata is protected.

Fig. 3.1(a) shows the initial state of the main Git repository, which contains two branches, “master” and “feature.” The local repository of developer 1 is in the same state as shown in

Fig. 3.1(a). The "feature" branch implements a new feature and contains one commit, C2. The code in C2 corresponds to an unstable, potentially-vulnerable version that needs to be tested more thoroughly before being integrated into the master branch. Commit C1 is the head of the master branch. This means that the file `.git/refs/heads/master` contains the SHA1 hash of the C1 commit object.

After developer 2 pulls from the master branch of the main repository (Fig. 3.1(b)), the attacker changes the master branch to point to commit C2 (Fig. 3.1(c). The attacker does this by simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C2 commit. Any developer who clones the repository or fetches from the master branch at this point in time will be provided with the incorrect repository state, as shown in Fig. 3.1(c). For example, developer 2, who committed C3 into his local repository (Fig. 3.1(d)), now wants to push this change to the main repository. Developer 2 is notified that there were changes on the master branch since his last fetch, and needs to pull these changes. As a result, a merge commit C4 occurs between C3 and C2 in the local repository of developer 2, as shown in Fig. 3.1(e). The main repository looks like Fig 3.1(e) after developer 2 pushes his changes. If developer 1 then pulls changes from the main repository, all three repositories will appear like Fig 3.1(e).

Normally, the master branch should contain software that was thoroughly tested and properly audited. However, in this incorrect history, the master branch incorporates commit C2, which was in a experimental feature branch and may contain bugs. The attacker tricked a developer into performing an action that was never intended, and none of the two developers are aware that the attack took place.

Tag Teleport Attacks. These attacks modify a tag reference so that it points to an arbitrary tag object. Surprisingly, a tag reference can also be made to point to a commit object, and Git commands will still work.

One can verify whether a tag is both signed and a valid tag object by using the `git tag --verify` command. However, if an attacker were to modify a tag reference to point to an older tag (e.g., if the tag for release 1.1 is replaced by the tag for the vulnerable release 1.0), the verification command is successful.

Modifying tag metadata could be especially impactful for automated systems that rely on tags to build/test and release versions of software [106, 132, 164, 237]. Furthermore, package managers such as Python's pip, Ruby's RubyGEMS, and Node's NPM, among many others support the installation of software from public Git repositories and tags. Finally, Git submodules are also vulnerable if used with the `--remote` parameter, as they track a remote tag (or branch). If a build dependency is included in a project as a part of the submodule, a package might be made vulnerable via an underlying library.

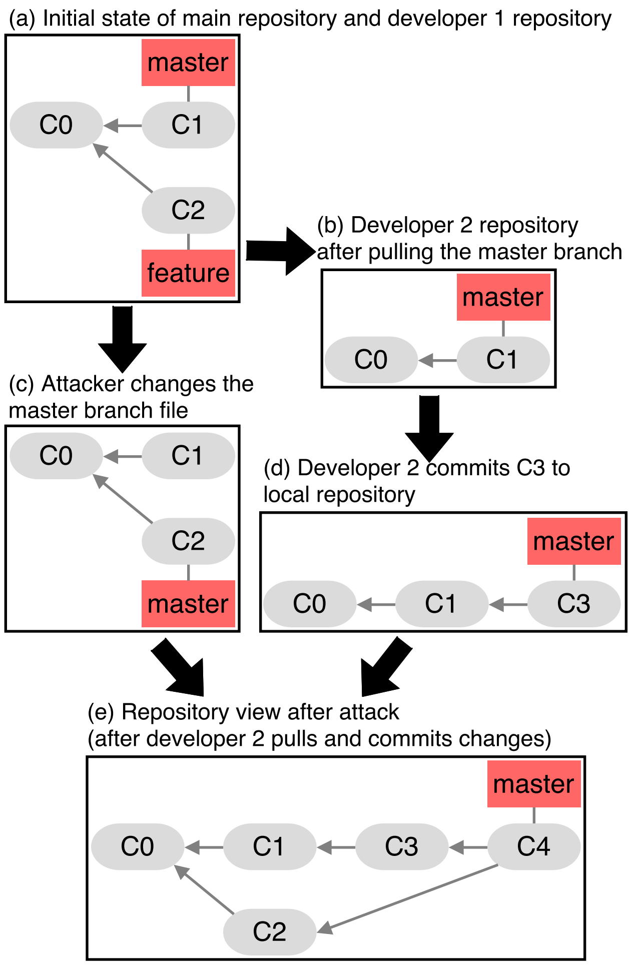

(a) Initial state of main repository and developer 1 repository
master
C0
C1
C2
feature
(b) Developer 2 repository after pulling the master branch
master
C0
C1
(c) Attacker changes the master branch file
C0
C1
C2
master
(d) Developer 2 commits C3 to local repository
master
C0
C1
C3
(e) Repository view after attack (after developer 2 pulls and commits changes)
master
C0
C1
C3
C4
C2

Figure 3.1: The Branch Teleport attack

3.4.2 Rollback Attacks

These attacks modify a Git branch reference so that it points to an older commit object from the same branch. This gives clients a view in which one or more of the latest branch commits are missing. The attacker can cause commits to be missing on a permanent or on a temporary basis.

Permanent rollback attacks

Based on the nature of the commits removed, we separate permanent rollback attacks in two groups: Branch Rollback attacks and Global Rollback attacks.

Branch Rollback Attacks. Consider the repository shown in Fig. 3.2(a), in which the order of the commits is C0, C1, C2, C3. Commits C0 and C3 are in the master branch, and commits C1 and C2 are security patches in a "patch" branch. The attacker rolls back the patch branch by

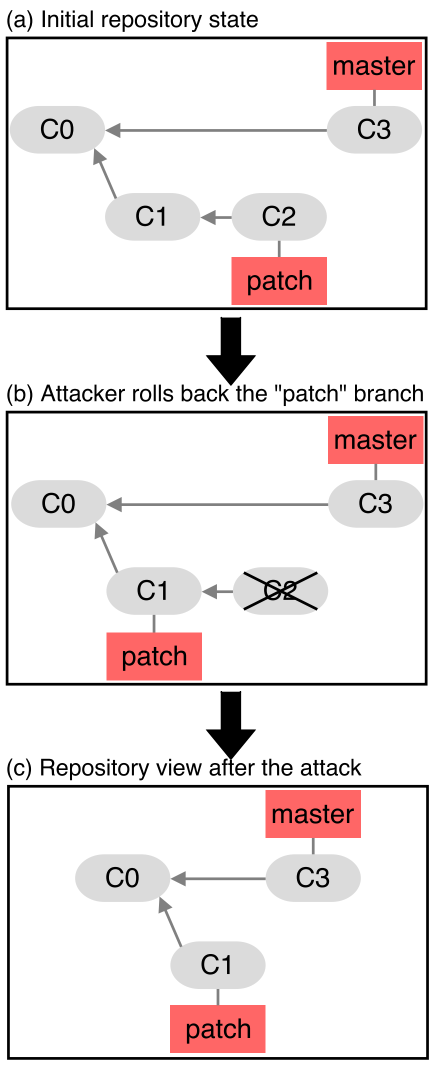

(a) Initial repository state
master
C0
C3
C1
C2
patch
(b) Attacker rolls back the "patch" branch
master
C0
C3
C1
C2
patch
(c) Repository view after the attack
master
C0
C3
C1
patch

Figure 3.2: The Branch Rollback attack

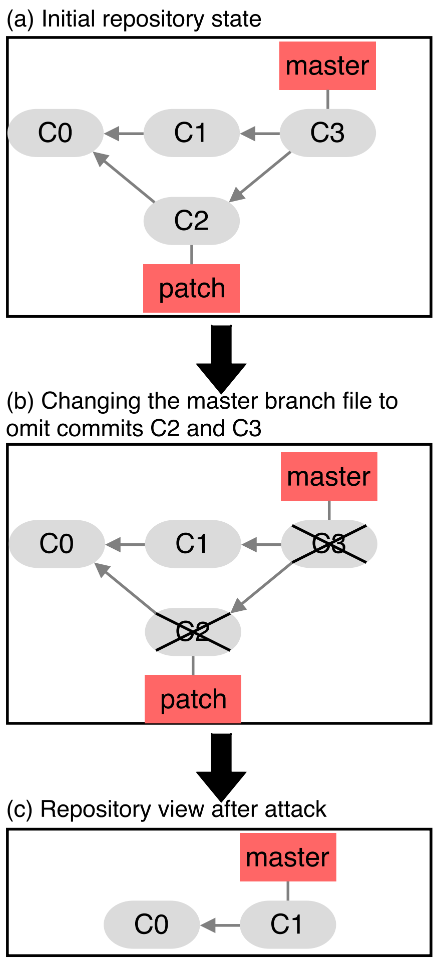

(a) Initial repository state
master
C0
C1
C3
C2
patch
(b) Changing the master branch file to omit commits C2 and C3
master
C0
C1
C3
C2
patch
(c) Repository view after attack
master
C0
C1

Figure 3.3: The Global Rollback attack

making the head of such branch point to commit C1, as shown in Fig. 3.2(b). This can be done by simply replacing the contents of the file `.git/refs/heads/patch` with the SHA1 hash of the C1 commit. As a result, all developers that pull from the main repository after this attack will see the state shown in Fig. 3.2(c), in which commit C2 (that contains a security patch) has been omitted.

Note that the attack can also be used to omit commits on any branch, including commits in the master branch.

Global Rollback Attacks. As opposed to a Branch Rollback attack, which removes commits that happened prior to one that remains visible, in a Global Rollback attack, no commits remain visible after the commits that are removed. In other words, the attacker removes one or more

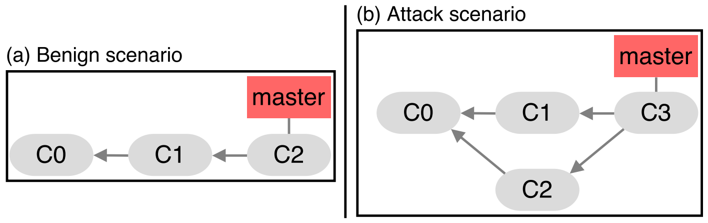


Figure 3.4: The Effort Duplication attack

commits that were added last to the repository.

Consider the initial state of a Git repository as illustrated in Fig. 3.3(a), in which C2 is a commit that fixes a security bug and has been merged into the master branch. The file `.git/refs/heads/master` contains the SHA1 hash of the C3 commit object.

By simply changing the contents of the file `.git/refs/heads/master` to the SHA1 hash of the C1 commit, the attacker forges a state in which the repository contains the history of commits depicted in Fig. 3.3(b). This effectively removes commits C2 and C3 from the project's history, and a developer who now clones the project will get a history of commits as shown in Fig. 3.3(c). This incorrect history does not contain the commit C2 that fixed the security bug.

Note that the Global Rollback attack *removed the latest two commits from the repository*. This is different than the effect of a Branch Rollback attack which *removes one or more commits that happened before a commit that remains visible*.

Temporary rollback attacks

Effort Duplication Attacks. The Effort Duplication attack is a variation of the Global Rollback attack, in which the attacker temporarily removes commits from the repository. This might cause developers to unknowingly duplicate coding efforts that exist in the removed commits.

Consider a main Git repository with just a master branch which contains only one commit C0. Two developers D1 and D2 have pulled from the main repository, so their local repositories also contain C0. After the following sequence of actions by D1 and D2, the repository should look as shown in Fig. 3.4(a):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository.
3. D2 commits C2 to her local repository & pushes to the main repository.

However, when D2 pulls in step 2, the attacker can temporarily withhold commit C1, keeping D2 unaware of the changes in C1. As a result, D2 works on changes that already exist in C1. The following attack scenario results in a repository shown in Fig. 3.4(b):

1. D1 commits C1 to her local repository & pushes to the main repository.
2. D2 pulls from the main repository, but the attacker withholds C1. Thus, D2 thinks there are no changes.
3. D2 makes changes on top of C0 and commits these changes in her local repository as commit C2. C2 duplicates (some or all of) D1's coding effort in C1.
4. D2 tries to push changes to the main repository. This time, the attacker presents C1 to D2 (these are the changes that were withheld in step 2). Thus, D2 has to first pull changes before pushing.
5. D2 pulls changes from the main repository, and this results in a merge commit C3 between C1 and C2. As part of the merge, the developer has to solve any merge conflicts that appear from the code duplication between C1 and C2.

In this case, D2 re-did a lot of D1's work because D1's commit C1 was withheld by the attacker. Note that unlike a Global Rollback attack, in which commits are removed permanently from the repository, in the Effort Duplication attack commits are just removed temporarily. This is a more subtle attack, since the final state of the repository is the same for both the benign and attack cases. The effect of applying commits C1 and C2 in Fig. 3.4(a) on the files in the repository is the same as applying commits C1, C2, C3 in Fig. 3.4(b). However, D2 unknowingly (and unnecessarily) duplicated D1's coding effort, which may have negative economic consequences. Adding to this, an attacker can slow down developers of a specific project (e.g., a competitor's project) by delivering previously-withheld changes to them when they will cause merge conflicts and hamper their development progress.

3.4.3 Reference Deletion Attacks

Since the branch metadata is not protected, the attacker can hide an entire branch from the repository by removing a branch reference. Similarly, since the tag metadata is not protected, the attacker can remove a tag reference in order to hide a release from the repository history.

When an attacker performs a reference deletion attack, only the users who previously held a copy of the reference will be able to know of its existence. If this is not the case, a developer would be oblivious of the fact that other developers have worked on the deleted branch (similar to a fork attack), or be tricked into retrieving another version if the target tag is not available. Furthermore, some projects track work in progress by tying branch names to numbers in their issue tracker [105], so two developers could be tricked into working on the same issue by hiding a branch (similar to an effort duplication attack).

3.4.4 Summary of Attacks

Metadata manipulation attacks may lead to inconsistent and incorrect views of the repository and also to corruption and loss of data. Ultimately, this will lead to merge conflicts, omission of bug fixes, merging experimental code into a production branch, or withholding changes from certain users before a release. All of these are problems that can impact the security and stability of the system as a whole. Table 3.1 summarizes the attacks impact.

Attack	Impact
Branch Teleport	Buggy code inclusion
Branch Rollback	Critical code omission
Global Rollback	Critical code omission
Effort Duplication	Coding effort duplicated
Tag Rollback	Older version retrieved

Table 3.1: Impact of metadata manipulation attacks.

3.5 Defense Framework

3.5.1 Design Goals for a Defense Scheme

We designed our defense scheme against metadata manipulation attacks with the following goals in mind:

*Design Goal 1 (**DG1**):* Achieve the security goals stated in Sec. 3.3.1. That is, prevent modification of committed data, ensure a consistent repository state, and ensure repository state freshness.

*Design Goal 2 (**DG2**):* Preserve (as much as possible) current workflows and actions that are commonly used by developers, in order to facilitate a seamless adoption.

*Design Goal 3 (**DG3**):* Maintain compatibility with existing Git implementations. For example, Git has limited functionality when dealing with concurrency issues in a multi-user setting: it only allows atomic push of multiple branches and tags after version 2.4. Following Git's design philosophy, backwards compatibility is paramount; a server running the latest Git version (*i.e.*, 2.9.0) can be cloned by a client with version 1.7.

3.5.2 Why Binding References with Git Objects is not Enough

Adding reference information (i.e., branch and tag names) inside the commit object might seem like a sufficient defense against metadata manipulation attacks. This would bind a commit to a reference and prevent an attacker from claiming that a commit object referred to in a reference belongs somewhere else.

Unfortunately, this simple approach has important drawbacks. It does not meet our *DG1* because it does not defend against rollback and effort duplication attacks. Furthermore, adding new reference information in a commit object requires updating an existing commit object. When this happens, the SHA-1 hash of the commit object will change, and the change will propagate to all new objects in the history. In other words, when a new branch is created and bound to a commit far earlier in the history, all commit objects need to be rewritten and, thus, sent back to the remote repository, which could add substantial computational and network overhead.

3.5.3 Our Defense Scheme

The fundamental cause of metadata manipulation attacks is that the server can respond to users' `fetches` with an incorrect state and history of the main repository that they cannot verify. For example, the server can falsely claim that a branch points to a commit that was never on that branch or to a commit that was the location of that branch in an earlier version. Or, the server can falsely claim that the reference of a tag object points to an older tag.

In order to stop the server from falsely claiming an incorrect state of the repository, we propose that every Git user must include additional information vouching for their perceived repository state during a push or a fetch operation. To achieve this, we include two pieces of additional information on the repository:

- First, upon every push, users must append a *push entry* to a *Reference State Log (RSL)*. By validating new entries in this log with each push and fetch operation, we can prevent teleport, permanent rollback, and deletion attacks.

- Second, when a Git user performs a fetch operation and receives a new version of files from the repository, the user places a random value into a *fetch nonce bag*. If the Git user does not receive file updates when fetching, the user replaces her value in the bag with a new one. The bag serves to protect against temporary rollback attacks.

During our descriptions, we assume that a trusted keychain is distributed among all developers along with the RSL. There are tools available to automate this process [85, 170], and the RSL itself can also be used to distribute trust (we elaborate more on this in Sec 3.6.1).

The Reference State Log (RSL)

For a developer to prevent the server from equivocating on the location of the references, the developer will sign a *push entry*, vouching for the location of the references at the time of a

PROCEDURE: Secure_push
Input: LocalRSL; related_commits; pushed reference X
Output: *result: (success/fail/invalid)*

```
 1: repeat
 2:     $result \leftarrow$ fail
 3:     (RemoteRSL, nonce_bag) =
            Retrieve_RSL_and_nonce_bag_from_remote_repo
 4:     if (RSL_Validate(RemoteRSL, nonce_bag) == false) then
 5:         // Retrieved RemoteRSL is invalid
 6:         // Must take necessary actions!
 7:         return invalid
 8:     end if
 9:     if (new push entries for reference X in RemoteRSL) then
10:         // Remote repository contains changes
11:         // User must fetch changes and then retry
12:         return fail
13:     else
14:         prev_hash = hash_last_push_entry(RemoteRSL)
15:         new_RSL_Entry = create_push_Entry(prev_hash,
                                               nonce_bag, X)
16:         nonce_bag.clear()
17:         RemoteRSL.addEntry(new_RSL_entry)
18:         result = Store_in_remote_repo(RemoteRSL,
                                          nonce_bag)
19:         if ($result$ == success) then
20:             // The remote RSL has no new entries
21:             push related_commits
22:             LocalRSL = RemoteRSL
23:             return success
24:         end if
25:     end if
26: until ($result$ == success)
```

push. To do this, she must execute the Secure_push procedure, which has the following steps:

First, the remote RSL is retrieved, validated, and checked for the presence of new push entries (lines 3-11). If the RSL is valid and no push entries were added, a new RSL push entry is created (lines 13-14). The newly created entry will contain: (1) the new location of the reference being pushed; (2) the nonces from the fetch nonce bag; (3) a hash of the previous push entry; and (4) the developer's signature over the newly created push entry. The newly created entry is then appended to the RSL (line 16), and the nonce_bag is cleared (line 15).

Once this is done, the remote RSL must be updated and local changes must be pushed to the remote repository (lines 17-20). Notice that these steps are performed under a loop, because other developers might be pushing, which is not an atomic operation in older versions of Git (this is required to meet *DG3*).

Depending on the result of the Secure_push procedure, a developer's actions correspond to the following:

- `success`: the push is successful. No further actions are required from the user (line 22).
- `fail`: the push fails because there are changes in the remote repository that must first be fetched and merged locally before the user's changes can be pushed (line 11).
- `invalid`: the RSL validation has failed. The algorithm detects a potential attack and notifies the user, who must then take appropriate measures (line 7).

Note that these actions mirror a user's actions in the case of a regular Git push operation, as suggested by *DG2*. By doing this, we effectively follow the existing Git workflows while providing better security guarantees at the same time.

The Nonce Bag

When retrieving the changes from a remote repository, a developer must also record her perceived state of the repository. Our scheme requires that all the user fetches be recorded in the form of a fetch *nonce bag*, *i.e.*, an unordered list of nonces. Each nonce is a random number that corresponds to a fetch from the main repository. Every time a user fetches from the main repository, she updates the nonce bag. If the user has not fetched since the last push, then she generates a new nonce and adds it to the nonce bag; otherwise, the user replaces her nonce in the nonce bag with a new nonce.

Each nonce in the nonce bag serves as a proof that a user was presented a certain RSL, preventing the server from executing an Effort Duplication attack and providing repository freshness as per *DG1*. To fetch the changes from the remote repository, a developer must execute the Secure_fetch procedure.

The first steps of the Secure_fetch procedure consist of retrieving the remote RSL, performing a regular `git fetch`, and ensuring that the latest push entry in the RSL points to a valid object in the newly-fetched reference (lines 4-11). Note that this check is performed inside a loop because push operations are not atomic in older versions of Git (lines 2-12). A user only needs to retrieve the entries which are new in the remote RSL and are not present in the local version of the RSL.

If this check is successful, the nonce bag must be updated and stored at the remote repository (lines 14-20). Note that all these steps are also in a loop because other developers might update the RSL or the nonce bag since it was last retrieved (lines 1-21).

Finally, the RSL is further validated for consistency (line 22), and the local RSL is updated. We chose to validate the RSL at the end of Secure_fetch and outside of the loop in order to optimize for the most common case. Once Secure_fetch is successfully executed, a developer can

PROCEDURE: Secure_fetch
Input: reference X to be fetched
Output: *result: (success/invalid)*

```
repeat
    $store\_success \leftarrow$ false
    repeat
        (RemoteRSL, nonce_bag) =
            Retrieve_RSL_and_nonce_bag_from_remote_repo()
        $fetch\_success \leftarrow$ false
        // This is a regular "Git fetch" command.
        // Branch X's reference is copied to FETCH_HEAD
        fetch reference X
        C ← RemoteRSL.latestPush(X).refPointer
        if (C == FETCH_HEAD) then
            $fetch\_success \leftarrow$ true
        end if
    until ($fetch\_success ==$ true)
    // Update the nonce bag
    if NONCE in nonce_bag then
        nonce_bag.remove(NONCE)
    end if
    save_random_nonce_locally(NONCE)
    nonce_bag.add(NONCE)
    // Storing the nonce bag at the remote repository
    //  might fail due to concurrency issues
    $store\_success$ = Store_in_remote_repo(nonce_bag)
until ($store\_success ==$ true)
if (RSL_Validate(RemoteRSL, nonce_bag) == false) then
    // Retrieved RemoteRSL is invalid
    // Must take necessary actions!
    return invalid
else
    LocalRSL = RemoteRSL
    return success
end if
```

be confident that the state of the repository she fetched is consistent with her peers. Otherwise, the user could be the victim of one of the attacks in Sec. 3.4.

RSL validation

The RSL_Validate routine is used in Secure_push and Secure_fetch to ensure the presented RSL is valid. The aim of this routine is to check that push entries in a given RSL are correctly linked to each other, that they are signed by trusted developers, and that nonces corresponding to a user's fetches are correctly incorporated into the RSL.

First, the procedure checks that the nonce corresponding to the user's last fetch appears either in the nonce bag or was incorporated into the right push entry (*i.e.*, the first new push entry of the remote RSL) (lines 1-2). The algorithm then checks if the new push RSL entries from the remote RSL are correctly linked to each other and that the first new remote push entry is correctly linked to the last push entry of the local RSL (the check is based on the prev_hash field) (lines 5-9). Finally, the signature on the last RSL push entry is verified to ensure it was signed by a trusted developer; since all RSL entries are correctly linked, only the last entry signature needs to be verified.

PROCEDURE: RSL_Validate
Input: LocalRSL (RSL in the local repository); RemoteRSL; nonce_bag
Output: `true` or `false`

```
 1: if (NONCE not in nonce_bag) and (NONCE not in RemoteRSL.push_after(LocalRSL) then
 2:     return false
 3: end if
 4: // Verify that the ensuing entries are valid
 5: prev_hash = hash_last_push_entry(LocalRSL)
 6: for new_push_entry in RemoteRSL do
 7:     if new_push_entry.prev_hash != prev_hash then
 8:         // The RSL entries are not linked correctly
 9:         return false
10:     end if
11:     prev_hash = hash(new_push_entry)
12: end for
13: if verify_signature(RemoteRSL.latest_push) == false then
14:     // this RSL is not signed by a trusted developer
15:     return false
16: end if
17: return true
```

How to handle misbehavior? If the RSL validation fails due to a misbehaving server, the user should compare the local RSL with the remote RSL retrieved from the remote repository and determine a safe point up to which the two are consistent. The users will then manually roll back the local and remote repositories to that safe point, and decide whether or not to continue trusting the remote repository.

3.6 Discussion

3.6.1 Trust and Revoke Entries

Developers' keys may be distributed using *trust/revoke RSL entries*. To use these entries, the repository is initialized with an authoritative root of trust (usually a core developer) who will add further entries of new developers in the group. Once developers' public keys are added to the RSL, they are allowed to add other trust entries.

A trust entry contains information about the new developer (i.e., username and email), her public key, a hash of the previous push entry and a signature of the entry by a trusted developer. Revocation entries are similar in that they contain the key-id of the untrusted developer, the hash of the push entry, and the signature of the developer revoking trust.

	Possible attacks	Time window of attack	Vulnerable commit objects
Commit signing	all attacks	Anytime	Any object
RSL (full adoption)	no attacks	None	No object
RSL (partial adoption)	all attacks	After the latest RSL entry and before the next RSL entry	Objects added after the latest RSL entry

Table 3.2: Security guarantees offered by different adoption levels of the defense scheme

3.6.2 Security Analysis

Our defense scheme fulfills the properties described in Sec. 3.3.1 as follows:

- Prevent modification of committed data: The existing signing mechanism for Git handles this well. Also, RSL entries are digitally signed and chained with each other, so unauthorized modifications will be detected.
- Ensure consistent repository state: The RSL provides a consistent view of the repository that is shared by all developers.
- Ensure repository state freshness: The Nonce Bag provides repository state freshness because an attacker cannot replay nonces in the Nonce Bag. Also, if no newer push entries are provided by the repository, then the attack becomes a fork attack.

The attacks described in Section 3.4 are prevented because, after performing the attack, the server cannot provide a valid RSL that matches the current repository state. Since she does not control any of the developers' keys, she can not forge a signature for a spurious RSL entry. As a result, a user who fetches from the main repository after the attack will notice the discrepancy between the RSL and the repository state that was presented to her. Each metadata manipulation attack would be detected as follows:

- *Branch Teleport and Deletion Attacks:* When this attack is performed, there is no mention of this branch pointing to such a commit, and the RSL validation procedure will fail.
- *Branch/Global Rollback and Tag Teleport Attacks:* These attacks can be detected because the latest entry in the RSL that corresponds to that branch points to the commit removed and the RSL validation procedure will fail.

  An attacker can attempt to remove the latest entries on the RSL so that the attacks remain undetected. However, after this moment, the server would need to consistently provide an incorrect view of the RSL to the target user, which would result in a fork attack. Finally, the attacker cannot remove RSL entries in between because these entries are chained using the previous hash field. Thus, the signature verification would fail if this field is modified.

- *Effort Duplication Attack:* This attack will result in a fork attack because the RSL created by the user requesting the commit will contain a proof about this request in the form of a nonce that has been incorporated into an RSL push entry or is still in the Nonce Bag. Any ensuing RSL push entry that was withheld from the user will not contain the user's nonce.

3.6.3 Partial Adoption of Defense Scheme

It is possible that not all developers in a Git repository use our solution. This can happen when, for example, a user has not configured the Git client to sign and update the RSL. When this is the case, the security properties of the RSL change.

To study the properties of using the RSL when not everyone is using the defense, we will define a commit object as a *"secure commit"* or an *"insecure commit."* The former will be commits made by users who employ our defense, while the latter are made by users who do not use our defense (*i.e.*, they only use the Git commit signing mechanism). Consider that supporting partial adoption requires changing the validation during fetches to consider commits that are descendants of the latest secure commit, for users might push to branches without using the defense. For simplicity, we do not allow users to reset branches if they are not using the defense.

Compared to commit signing only, when our scheme is adopted by only some users, a user who writes an RSL entry might unwittingly attest to the insecure commits made by other users after the latest secure commit. However, this situation still provides a valuable advantage because the attacker's window to execute Metadata Manipulation attacks is limited in time. That is, when our defense is not used at all, an attacker can execute Metadata Manipulation attacks on any commits in the repository, (e.g., the attacker can target a forgotten branch located early in the history). This is not possible with our scheme, where an attacker can only attack the commit objects added after the latest RSL entry for that branch. The differences between the three alternatives are summarized in Table 3.2.

3.6.4 Comparison with Other Defenses

In Table 3.3, we examine the protections offered by other defense schemes against metadata manipulation attacks. Specifically, we studied how Git commit signing, Git's push certificate solution, and our solution (listed as RSL) fare against the attacks presented in this chapter, as well as other usability aspects that may impact adoption.

Feature	**Commit signing**	**Push certificate**	**RSL**
Commit Tampering	✓	✓	✓
Branch Teleport	X	✓	✓
Branch Rollback	X	X	✓
Global Rollback	X	X	✓
Effort Duplication	X	X	✓
Tag Rollback	X	✓	✓
Minimum Git Version	1.7.9	2.2.0	1.7.9
Distribution Mechanism	in-band	(no default) or Additional server	in-band

Table 3.3: Comparison of defense schemes against Git metadata manipulation attacks. A ✓indicates the attack is prevented.

As we can see, Git commit signing does not protect against the vast majority of attacks presented in this chapter. Also, Git's push certificate solution provides a greater degree of protection, but still fails to protect against all rollback and effort duplication attacks. This is primarily because (1) a server could misbehave and not provide the certificates (there is no default distribution mechanism), and (2) a server can replay old push certificates along with an old history. Basically, this solution assumes a well-behaving server hosting push certificates.

In contrast, our solution protects against all attacks presented in Table 3.3. In addition to this, our solution presents an in-band distribution mechanism that does not rely on a trusted server in the same way that commit signing does. Lastly, we can see that our solution can be used today, because it does not require newer versions of Git on the client and requires no changes on the server, which allows for deployment in popular Git hosting platforms such as GitHub and GitLab.

3.7 Implementation and Evaluation

We have implemented a prototype for our defense scheme. This section provides implementation details and presents our experimental performance results.

3.7.1 Implementation

To implement our defense scheme, we leveraged *Git custom commands* to replace the push and fetch commands, and implemented the RSL as a separate branch inside the repository itself. To start using the defense, a user is only required to install two additional bash scripts and use them in lieu of the regular fetch and push commands. Our client scripts consist of less than a hundred (86) lines of code, and there is no need to install anything on the server.

RSL and Nonce Bag. We implemented the RSL in a detached branch of the repository, named "RSL." Each RSL entry is stored as a Git commit object, with the entry's information encoded in the commit message. We store each entry in a separate commit object to leverage Git's pack protocol, which only sends objects if they are missing in the local client. Encoding the Git commit objects is also convenient because computing the previous hash field is done automatically.

We also represent the Nonce Bag as a Git commit object at the head of the RSL branch. When a nonce is added or updated, a new commit object with the nonces replaces the previous nonce bag, and its parent is set to the latest RSL entry. When a new RSL entry is added, the commit containing the nonce bag is garbage collected by Git because the RSL branch cannot reach it anymore.

Field	Description
Branch	Target branch name
HEAD	Branch location (target commit)
PREV_HASH	Hash over the previous RSL entry
Signature	Digital signature over RSL entry

Table 3.4: RSL push entry fields.

When `securepush` is executed, the script first fetches and verifies the remote RSL branch. If verification is successful, it then creates an RSL entry by issuing a new commit object with a NULL tree (i.e., no local files), and a message consisting of the fields described in Table 3.4. After the new commit object with the RSL push entry is created, the RSL branch is pushed to the remote repository along with the target branch.

A `securefetch` invocation will fetch the RSL branch to update or add the random nonce in the Nonce Bag. If a nonce was already added to the commit object (with a NULL tree also), it will be amended with the replaced nonce. In order to keep track of the nonce and the commit object to which it belongs, two files are stored locally: NONCE_HEAD, which contains the reference of the Nonce Bag in the RSL branch, and NONCE, which contains the value of the nonce stored in it.

Atomicity of Git operations. The *securepush* and *securefetch* operations require fetching and/or pushing of the RSL branch in addition to the pushing/fetching to/from the target branch. Git does not support atomic fetch of multiple branches, and only supports atomic push of multiple branches after version 2.4.0 [1].

In order to ensure backwards compatibility, we designed our solution without considering the existence of atomic operations. Unfortunately, the lack of atomic push opens the possibility of a DoS attack that exploits the 'repeat' loop in Secure_fetch (lines 3-12), that makes the algorithm loop endlessly. This could happen if a user executes Secure_push and is interrupted after pushing a new RSL entry, but before pushing the target branch (e.g., caused by a network failure). Also, a malicious user may provide an updated RSL, but an outdated history for that branch. However, this issue can be easily solved if the loop is set to be repeated only a finite number of times before notifying the user of a potential DoS attempt.

If atomic push for multiple branches is available, the Secure_push procedure can be simplified by replacing lines 17-22 with a single push. Availability of atomic push also eliminates the possibility of the endless loop mentioned above.

3.7.2 Experimental Evaluation

Experimental Setup. We conducted experiments using a local Git client and the GitHub server that hosted the main repository. The client was running on an Intel Core i7 system with two CPUs and 8 GB RAM. The client software consisted of OS X 10.11.2, with Git 2.6.2 and the GnuPG 2.1.10 library for 1024-bit DSA signatures.

Our goal was to evaluate the overhead introduced by our defense scheme. Specifically, we want to determine the additional storage induced by the RSL, and the additional end-to-end delay induced by our `securefetch` and `secure push` operations. For this, we used the five most popular GitHub repositories [2]: bootstrap, angular.js, d3, jQuery, oh-my-zsh. We will refer to these as R1, R2, R3, R4, and R5, respectively. In the experiments, we only considered the commits in the master branch of the these repositories. Table 3.5 provides details about these repositories.

We used the repositories with signed commits as the baseline for the evaluation. We evaluated three defense schemes:

- Our defense: This is our proposed defense scheme.
- Our defense (light): A light version of our defense scheme, which does not use the nonce bag to keep track of user fetches. This scheme sacrifices protection against Effort Duplication attacks in favor of keeping the regular Git `fetch` operation unchanged.

[1]Note that **both Git client and server** must be at least version 2.4.0 in order to support atomic push.

[2]Popularity is based on the "star" ranking used by GitHub, which reflects users' level of interest in a project (retrieved on Feb 14, 2016).

Repo.	Number of commits	Number of pushes	Repo. size	Repo. size with signed commits
R1	11,666	1,345	73.04	78.85
R2	7,521	26	66.96	69.79
R3	3,510	255	32.91	34.65
R4	6,031	194	15.79	19.98
R5	3,841	1,170	3.52	4.01

Table 3.5: The repositories used for evaluation (sizes are in MBs).

- Push certificates: Push certificates used upon pushing.

For our defense and our defense (light), the repositories were hosted on GitHub. Given that GitHub does not support push certificates, we studied the network overhead using a self-hosted server on an AWS instance, and concluded that push certificates incur a negligible overhead compared to the baseline. Thus, we only compare our scheme with push certificates in regard to the storage overhead.

Storage overhead. Table 3.6 shows the additional storage induced by our defense, compared to push certificates. In our defense, the RSL determines the size of the additional storage. We can see that our defense requires between 0.009%-6.5% of the repository size, whereas push certificates require between 0.012%-10%. The reason behind this is that push certificates contain 7 fields in addition to the signature, whereas RSL push entries only have 3 additional fields.

Repo.	Our defense	Push certificates
R1	301.93	461.27
R2	6.49	8.88
R3	58.91	86.05
R4	44.34	66.27
R5	261.3	402.19

Table 3.6: Repository storage overhead of defense schemes (in KBs).

Communication overhead. To evaluate the additional network communication cost introduced by our `securepush` operation when compared to the regular `push` operation, we measured the cost of the last 10 pushes for the five considered repositories. To evaluate the cost of `securefetch`, we measured the cost of a fetch after each of the last 10 pushes.

Table 3.7 shows the cost incurred by push operations. We can see that our defense incurs, on average, between 25.24 and 26.21 KB more than a regular `push`, whereas our defense (light) only adds between 10.29 and 10.48 KB. This is because a `securepush` in our defense retrieves, updates and then stores the RSL in the remote repository. In contrast, our defense (light) only requires storing the RSL with the new push entry if there are no conflicts. Table 3.8 shows the cost incurred by fetch operations. A `securefetch` incurs on average between 25.1 and 25.55 KB more than a regular `fetch`, whereas our defense (light) only adds between 14.34 and 10.91 KB.

The observed overhead is a consequence of the fact that we implemented our defense scheme to respect design goal DG3, (*i.e.* no requirement to modify the Git server software). Since we implemented the RSL and the Nonce Bag as objects in a separate Git branch, `secure-push` and `securefetch` require additional `push`/`fetch` commands to store/fetch these, and thus they incur additional TCP connections. Most of the communication overhead is caused by information that is automatically included by Git and is unrelated to our defense scheme. We found that Git adds to each `push` and `fetch` operation about 8-9 KBs of supported features and authentication parameters. If our defense is integrated into the Git software, the `securepush` and `securefetch` will only require one TCP session dramatically reducing the communication overhead. In fact, based on the size of an RSL entry ($\sim$325 bytes), which is the only additional information sent by a `securepush`/`securefetch` compared to a regular `push`/`fetch`, we estimate that the communication overhead of our defense will be less than 1KB per operation.

3.8 Conclusions

In this chapter, we present a new class of attacks against Git repositories. We show that, even when existing Git protection mechanisms such as Git commit signing, are used by developers, an attacker can still perform extremely impactful attacks, such as removing security patches, moving experimental features into production software, or causing a user to install a version of software with known vulnerabilities.

To counter this new class of attacks, we devised a backwards compatible solution that prevents metadata manipulation attacks while not obstructing regular Git usage scenarios. Our evaluation shows that our solution incurs less than 1% storage overhead when applied to popular Git repositories, such as the five most popular repositories in GitHub.

We performed responsible disclosure of these issues to the Git community. We have been working with them to address these issues. Some of our patches have already been accepted into Git version 2.9. We are continuing to work with the Git community to fix these problems.

Like we did with Git, it is paramount to ensure the state integrity, and the artifact exportability properties are provided by every step within the chain. This will require system designers and security professionals to consider cryptographic authentication of the state of each individual point, as well as integrity measures for the exportable artifacts for this step.

However, as we mentioned before, the individual security mechanisms of git will only matter if two things are true. First, that the consumers of the artifacts produced within this step are aware of the security policies in place. Second, that the consumers of the artifacts can verify their integrity. Thus, there there need be a higher-level mechanism that ensures these two properties hold. The resulting property when these are applied is third property named *artifact flow integrity*.

This property is the focus of `in-toto`, and it will be designed to allow consumers of artifacts to express the flow of artifacts and consumers to verify its existence. We will describe `in-toto` in depth next.

Scheme used	**R1**	**R2**	**R3**	**R4**	**R5**
Git w/ signed commits (baseline)	17.80	3,925.35	38.32	59.14	11.96
Our defense	44.01	3,950.87	63.56	84.71	37.65
Our defense (light)	28.28	3,935.71	48.61	69.52	22.28

Table 3.7: Average communication cost per push for the last 10 push operations, expressed in KBs.

Scheme used	**R1**	**R2**	**R3**	**R4**	**R5**
Git w/ signed commits (baseline)	20.68	3,896.98	40.93	65.85	13.67
Our defense	46.18	3,922.40	66.48	91.27	38.77
Our defense (light)	35.19	3,911.81	55.84	80.67	28.01

Table 3.8: Average communication cost per fetch for the last 10 fetch operations, expressed in KBs.

Scheme used	**R1**	**R2**	**R3**	**R4**	**R5**
Git w/ signed commits (baseline)	1.29	3.27	1.17	1.31	1.51
Our defense	3.11	5.27	2.78	2.95	3.51
Our defense (light)	2.44	4.49	2.16	2.40	2.81

Table 3.9: Average end-to-end delay per push for the last 10 push operations, expressed in seconds.

Scheme used	**R1**	**R2**	**R3**	**R4**	**R5**
Git w/ signed commits (baseline)	0.87	1.95	0.75	0.66	0.67
Our defense	2.93	3.86	2.52	2.40	2.75
Our defense (light)	1.60	2.75	1.52	1.31	1.30

Table 3.10: Average end-to-end delay per fetch for the last 10 fetch operations, expressed in seconds.

Chapter 4

in-toto: Holistic Software Supply Chain Security

4.1 Introduction

Modern software is built through a complex series of steps called a *software supply chain*. These steps are performed as the software is written, tested, built, packaged, localized, obfuscated, optimized, and distributed. In a typical software supply chain, these steps are "chained" together to transform (e.g., compilation) or verify the state (e.g., the code quality) of the project in order to drive it into a *delivered product*, i.e., the finished software that will be installed on a device. Usually, the software supply chain starts with the inclusion of code and other assets (icons, documentation, etc.) in a version control system. The software supply chain ends with the creation, testing and distribution of a delivered product.

Securing the supply chain is crucial to the overall security of a software product. An attacker who is able to control any step in this chain may be able to modify its output for malicious reasons that can range from introducing backdoors in the source code to including vulnerable libraries in the delivered product. Hence, attacks on the software supply chain are an impactful mechanism for an attacker to affect many users at once. Moreover, attacks against steps of the software supply chain are difficult to identify, as they misuse processes that are normally trusted.

Unfortunately, such attacks are common occurrences, have high impact, and have experienced a spike in recent years [126, 233]. Attackers have been able to infiltrate version control systems, including getting commit access to the Linux kernel [124] and Gentoo Linux [151], stealing Google's search engine code [56], and putting a backdoor in Juniper routers [110, 176]. Popular build systems, such as Fedora, have been breached when attackers were able to sign backdoored versions of security packages on two different occasions [149, 228]. In another prominent example, attackers infiltrated the build environment of the free computer-cleanup tool

CCleaner, and inserted a backdoor into a build that was downloaded over 2 million times [230]. Furthermore, attackers have used software updaters to launch attacks, with Microsoft [200], Adobe [175], Google [113, 148, 250], and Linux distributions [109, 254] all showing significant vulnerabilities. Perhaps most troubling are several attacks in which nation states have used software supply chain compromises to target their own citizens and political enemies [86, 98, 121, 173, 200, 208, 231, 232, 247]. There are dozens of other publicly disclosed instances of such attacks [22, 83, 91, 92, 95, 104, 117, 131, 140, 151, 156, 158, 160, 175, 199, 207, 215, 218, 220, 227, 238–240, 242, 248, 251, 266].

Currently, supply chain security strategies are limited to securing each individual step within it. For example, Git commit signing controls which developers can modify a repository [155], reproducible builds enables multiple parties to build software from source and verify they received the same result [65], and there are a myriad of security systems that protect software delivery [5, 50, 74, 182, 183]. These building blocks help to secure an individual step in the process.

Although the security of each individual step is critical, such efforts can be undone if attackers can modify the output of a step before it is fed to the next one in the chain [56, 111]. These piecemeal measures by themselves can not stop malicious actors because there is no mechanism to verify that: 1) the correct steps were followed and 2) that tampering did not occur in between steps. For example a web server compromise was enough to allow hackers to redirect user downloads to a modified Linux Mint disk image, even though every single package in the image was signed and the image checksums on the site did not match. Though this was a trivial compromise, it allowed attackers to build a hundred-host botnet in a couple of hours [266] due to the lack of verification on the tampered image.

In this chapter we introduce `in-toto`, Latin for "as a whole," the first framework that holistically enforces the integrity of a software supply chain by gathering cryptographically verifiable information about the chain itself. To achieve this, `in-toto` requires a project owner to declare and sign a *layout* of how the supply chain's steps need to be carried out, and by whom. When these steps are performed, the involved parties will record their actions and create a cryptographically signed statement — called *link metadata* — for the step they performed. The link metadata recorded from each step can be verified to ensure that all steps were carried out appropriately and by the correct party in the manner specified by the layout.

The layout and collection of link metadata tightly connect the inputs and outputs of the steps in such a chain, which ensures that tampering can not occur between steps. The layout file also defines requirements (e.g., Twistlock [73] must not indicate that any included libraries have high severity CVEs) that will be enforced to ensure the quality of the end product. These additions can take the form of either distinct commands that must be executed, or limitations on which files can be altered during that step (e.g., a step that localizes the software's documentation for Mexican Spanish must not alter the source code). Collectively, these requirements can minimize the impact of a malicious actor, drastically limiting the scope and range of actions such an attacker can perform, even if steps in the chain are compromised.

We have built a series of production-ready implementations of `in-toto` that have now been integrated across several vendors. This includes integration into cloud vendors such as Datadog and Control Plane, to protect more than 8,000 cloud deployments. Outside of the cloud, `in-toto` is used in Debian to verify packages were not tampered with as part of the reproducible builds project [65]. These deployments have helped us to refine and validate the flexibility and effectiveness of `in-toto`.

Finally, as shown by our security analysis of three `in-toto` deployments, `in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. Our `in-toto` deployments could be used to address most (between 83% - 100%) historical supply chain attacks.

4.2 Definitions and Threat Model

This section defines the terms we use to discuss the software supply chain and details the specific threat model `in-toto` was designed to defend against.

4.2.1 Definitions

The software supply chain refers to the series of *steps* performed in order to create and distribute a *delivered product*. A *step* is an operation within this chain that takes in *materials* (e.g., source code, icons, documentation, binaries, etc.) and and creates one or more *products* (e.g., libraries, software packages, file system images, installers, etc.). We refer to both materials and products generically as *artifacts*.

It is common to have the products of one step be used as materials in another step, but this does not mean that a supply chain is a sequential series of operations in practice. Depending on the specifics of a supply chain's workflow, steps may be executed in sequence, in parallel, or as a combination of both. Furthermore, steps may be carried out by any number of hosts, and many hosts can perform the same step (e.g., to test a step's reproducibility).

In addition to the materials and products, a step in the supply chain produces another key piece of information, *byproducts*. The step's byproducts are things like the `STDOUT`, `STDERR`, and return value that indicate whether a step was successful or had any problems. For example, a step that runs unit tests may return a non-zero code if one of the unit tests fails. Validating byproducts is key to ensuring that steps of the supply chain indicate that the software is ready to use.

As each step executes, information called *link* metadata that describes what occurred, is generated. This contains the materials, products, and byproducts for the step. This information is signed by a key used by the party who performs the action, which we call a *functionary*. Regardless of whether the functionary commits code, builds software, performs QA, localizes documentation, etc., the same link metadata structure is followed. Sometimes a functionary's participation involves repeated human action, such as a developer making a signed git commit for their latest code changes. In other cases, a functionary may participate in the supply chain in a nearly autonomous manner after setup, such as a CI/CD system. Further, many functionaries can be tasked to perform the same step for the sake of redundancy and a minimum threshold of them may be required to agree on the result of a step they all carried out.

To tie all of the pieces together, the *project owner* sets up the rules for the steps that should be performed in a software supply chain. In essence, the project owner serves as the foundation of trust, stating which steps should be performed by which functionaries, along with specifying rules for products, byproducts, and materials in a file called the *layout*. The layout enables a *client* that retrieves the software to cryptographically validate that all actions were performed correctly. In order to make this validation possible, a client is given the *delivered product*, which contains the software, layout, and link metadata. The layout also contains any additional actions besides the standard verification of the artifact rules to be performed by the client. These actions, called *inspections*, are used to validate software by further performing operations on the artifacts inside the delivered product (e.g., verifying no extraneous files are inside a zip file). This way, through standard verification and inspections, a client can assure that the software went through the appropriate software supply chain processes.

4.2.2 Threat Model

The goal of `in-toto` is to minimize the impact of a party that attempts to tamper with the software supply chain. More specifically, the goal is to retain the maximum amount of security that is practical, in any of the following scenarios:

- Interpose between two existing elements of the supply chain to change the input of a step. For example, an attacker may ask a hardware security module to sign a malicious copy of a package before it is added to the repository and signed repository metadata is created to index it [68, 101, 115, 151, 199, 260, 263, 263].
- Act as a step (e.g., compilation), perhaps by compromising or coercing the party that usually performs that step [23, 46, 68, 123, 130, 151, 152, 180, 264]. For example, a hacked compiler could insert malicious code into binaries it produces [230, 244].
- Provide a delivered product for which not all steps have been performed. Note that this can also be a result of an honest mistake [89, 112, 122, 135, 147, 177, 253].

- Include outdated or vulnerable elements in the supply chain [125,127,172,174,217]. For example, an attacker could bundle an outdated compression library that has many known exploits.

- Provide a counterfeit version of the delivered product to users [22,86,133,140,142,175, 218,242,243,266]. This software product can come from any source and be signed by any keys. While `in-toto` will not mandate how trust is bootstrapped, Section 4.6 will show how other protocols such as TUF [74], as well as popular package managers [5] can be used to bootstrap project owner keys.

Key Compromise. We assume that the public keys of project owners are known to the verifiers and that the attacker is not able to compromise the corresponding secret key. In addition, private keys of developers, CI systems and other infrastructure public keys are known to a project owner and their corresponding secret keys are not known to the attacker. In section 4.5.2, we explore additional threat models that result from different degrees of attacker access to the supply chain, including access to infrastructure and keys (both online and offline).

4.2.3 Security Goals

To build a secure software supply chain that can combat the aforementioned threats, we envision that the following security goals would need to be achieved:

- **supply chain layout integrity:** All of the steps defined in a supply chain are performed in the specified order. This means that no steps can be added or removed, and no steps can be reordered.

- **artifact flow integrity:** All of the artifacts created, transformed, and used by steps must not be altered in-between steps. This means that if step A creates a file `foo.txt` and step B uses it as a material, step B must use the exact file `foo.txt` created by step A. It must not use, for example, an earlier version of the file created in a prior run.

- **step authentication:** Steps can only be performed by the intended parties. No party can perform a step unless it is given explicit permission to do so. Further, no delivered products can be released unless all steps have been performed by the right party (e.g., no releases can be made without a signoff by a release engineer, which would stop accidental development releases [135]).

- **implementation transparency:** `in-toto` should not require existing supply chains to change their practices in order to secure them. However, `in-toto` can be used to represent the existing supply chain configuration and reason about its security practices.

- **graceful degradation of security properties:** in-toto should not lose all security properties in the event of key compromise. That is, even if certain supply chain steps are compromised, the security of the system is not completely undermined.

In addition to these security goals, in-toto is also geared towards practicality and, as such, it should maintain minimal operational, storage and network overheads.

4.3 System Overview

The current landscape of software supply chain security is focused on point-solutions that ensure that an individual step's actions have not been tampered with. This limitation usually leads to attackers compromising a weaker step in the chain (e.g., breaking into a build-farm [215]), removing steps from the chain [135] or tampering with artifacts while in transit (i.e., adding steps to the chain [133]). As such, we identify two fundamental limitations of current approaches to secure the software supply chain:

1. Point solutions designed to secure individual supply chain steps cannot guarantee the security of the entire chain as a whole.
2. Despite the widespread use of unit testing tools and analysis tools, like fuzzers and static analyzers, software rarely (if ever) includes information about what tools were run or their results. So point solutions, even if used, provide limited protection because information about these tools is not appropriately utilized or even shown to clients who can make decisions about the state of the product they are about to utilize.

We designed in-toto to address these limitations by ensuring that all individual measures are applied, and by the right party in a cryptographically verifiable fashion.

In concrete terms, in-toto is a framework to gather and verify metadata about different stages of the supply chain, from the first step (e.g., checking-in code on a version control system) to delivered product (e.g., a .deb installable package). If used within a software supply chain, in-toto ensures that the aforementioned security goals are achieved.

4.3.1 in-toto Parties and Their Roles

Similar to other modern security systems [181, 183, 222], in-toto uses security concepts like delegations and roles to limit the scope of key compromise and provide a graceful degradation of its security properties.

In the context of in-toto, a role is a set of duties and actions that an actor must perform. The use of delegations and roles not only provides an important security function (limiting the impact of compromise and providing separation of privilege), but it also helps the system remain flexible and usable so that behaviors like key sharing are not needed. Given that every

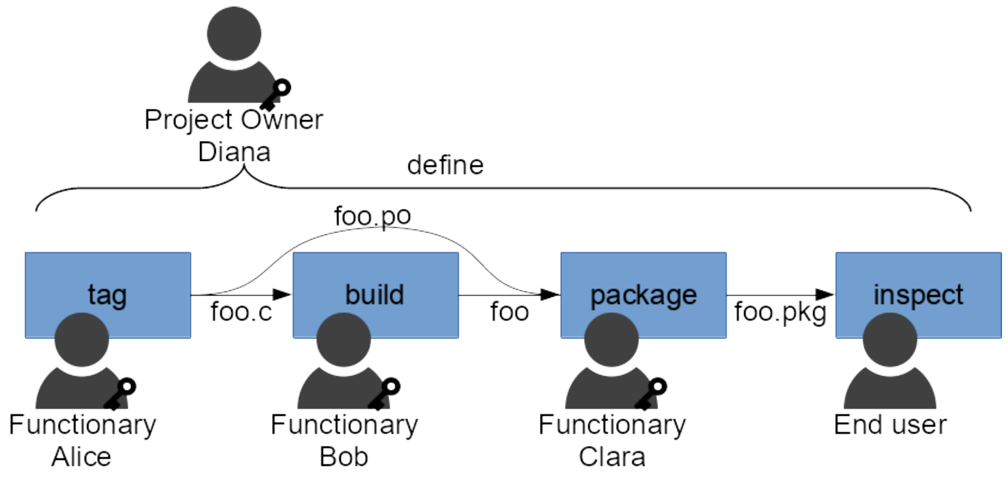


Figure 4.1: Graphical depiction of the software supply chain with `in-toto` elements added. The project owner creates a layout with three steps, each of which will be performed by a functionary.

project uses a very specific set of tools and practices, flexibility is a necessary requirement for `in-toto`. There are three roles in the framework:

- Project Owner: The project owner is the party in charge of defining the software supply chain layout (i.e., define which steps must be performed and by who). In practice, this would be the maintainer of an open-source project or the dev-ops engineers of a project.
- Functionaries: Functionaries are the parties that perform the steps within the supply chain, and provide an authenticated record of the artifacts used as materials and the resulting products. Functionaries can be humans carrying out a step (e.g., signing off a security audit) or an automated system (e.g., a build farm).
- Client: (e.g., end user): The client is the party that will inspect and afterwards utilize a delivered product.

We will now elaborate on how these three parties interact with the components of `in-toto`.

4.3.2 `in-toto` Components

`in-toto` secures the software supply chain by using three different types of information: the software supply chain layout (or layout, for short), link metadata, and the delivered product. Each of these has a unique function within `in-toto`.

The Supply Chain Layout

Laying out the structure of the supply chain allows the developers and maintainers of a project to define requirements for steps involved in source code writing, testing, and distribution within a software product's lifecycle. In the abstract sense, this supply chain layout is a recipe that identifies which steps will be performed, by whom, and in what order.

The supply chain layout defines a series of *steps* in the supply chain. These definitions are used to enforce measures on what artifacts should be used as *materials*. To ensure that only the intended parties execute the right steps, a public key is associated with each step. In order to ensure that the layout was created by the project owner, it is cryptographically signed with the project owner's private key.

The project owner will define this supply chain layout by setting different requirements for the project's steps. These requirements take the form of types of artifacts that can be produced (e.g., a localization step can only produce `.po` files), the expected return values, the type of host that can carry out this step and so forth. When consuming the delivered product, the client (end user) verifies that these requirements are satisfied.

In addition to defining supply chain steps, the layout will also specify a series of *inspection steps* (or inspections). These inspections will be performed by the verifier on the delivered product to draw further insight about its correctness. This is useful for complex supply chains in which the basic semantics of `in-toto` cannot describe their specific requirements. For example, an inspection step can be used to namespace restrict certain VCS-specific operations to specific functionaries such as making sure that only a QA team member merges code into the develop branch and that all commits are signed.

For example, as seen in Figure 4.1, a project owner can define a supply chain consisting of three steps: a tag, a build and a package step. With these definitions, the project owner also defines how the artifacts will flow through the supply chain (e.g., foo.c is used by build, yet foo.po is packaged directly from tag). Afterwards, the project owner can assign functionaries to carry out each of these steps and define an inspection so the end user can verify that foo was indeed created during build and that foo.po came from the tagged release.

Layout creation tool. We provide a web-based layout creation tool [34] to help project owners create `in-toto` layouts. The tool uses an intuitive, graphical interface to define: (1) the steps of the software supply chain (i.e., how is the source code managed? how is the software's quality verified? how is the software built? how is the software packaged?), (2) the actors (functionaries) who are allowed to perform different steps of the software supply chain. An `in-toto` layout is generated based on this information. In addition, the `in-toto` website [36,40] provides several examples of layouts, which can serve as starting templates for project owners seeking to integrate `in-toto`.

Link metadata

Verifying the actions carried out in the supply chain, requires information about all steps performed in creating the delivered product. Like a chain in real life, an `in-toto` supply chain consists of conjoined *links*, with each link serving as a statement that a given step was carried out.

Functionaries in charge of executing a step within the supply chain must share information about these links. Sharing such information as what *materials* were fed to the step, and what *product(s)* were created, can ensure no artifacts are altered in transit. To ensure that only the right functionaries performed this step, the piece of link metadata must be signed with the private key that corresponds to this functionary's key (as defined in the supply chain layout).

There is a one-to-one relationship between the step definitions in the supply chain layout and the link metadata. That is, each piece of link metadata gathered during each step within the supply chain must match what the requirements prescribe for that step. In order to ensure that the link metadata is generated by the intended entity, it must be cryptographically signed with one (or more, if there is a threshold higher than one defined) of the keys indicated in the requirements for that link.

When all the link metadata has been collected, and the supply chain has been properly defined, the supply chain layout and all the links can be shipped, along with the delivered product, to the end user for verification. We show a minimal software supply chain, along with a graphical representation of an `in-toto` layout in Figure 4.1.

The delivered product

The delivered product is the piece of software that the end user wants to install. In order to verify the delivered product, the end user (or client) will utilize the supply chain layout and its corresponding pieces of link metadata. The end user will use the link metadata to verify that the software provided has not been tampered with, and that all the steps were performed as the project owner intended. In Figure 4.1 the delivered product consists of the foo.pkg file.

4.3.3 `in-toto` Usage Lifecycle

The `in-toto` usage lifecycle encompasses the following overarching operations:

1. The project owner defines a supply-chain layout.
2. Each step is carried out as specified, and functionaries gather and sign link metadata.
3. A delivered product is shipped to the client, who verifies it upon installation by:
    1. ensuring the layout provided was signed by the project owner and is not expired.

2. checking that all the steps defined have enough pieces of link metadata; that such links were signed by the indicated functionaries; and that all artifacts recorded flowed properly between the steps as indicated in the layout.
3. carrying out any inspection steps contained in the layout and making sure that all artifacts recorded match the flow described in the layout.

As seen in Figure 4.1 a project owner creates the layout to describe an overarching structure of the supply chain that the client can use to verify. Later, functionaries carry out their operations as usual, and submit link metadata to attest for the result of their operation. Finally, a client uses a delivered product, metadata links and a layout to verify the integrity of the delivered product and of the entire chain.

By following the chain of attestations in the link metadata, the client can reconstruct the operations described in Figure 4.1. Afterwards, the client can verify these attestations against the layout and execute any inspections to make sure everything is in order before consuming the delivered product.

4.4 `in-toto` Internals

In order to avoid tampered, incomplete or counterfeit software, `in-toto` ensures the integrity and accuracy of all software supply chain operations. `in-toto` ensures supply chain integrity by the verifying the collected link metadata against a software supply chain layout file. This ensures that all operations were carried out, by the intended party and as the legitimate project owner intended.

Understanding how the system's metadata helps to ensure the integrity of the supply chain is critical to a deeper appreciation of how `in-toto` works. In this section, we will explore the specifics of the link metadata and the layout file to understand how `in-toto` operates.

For the context of this section, we will demonstrate the different features of `in-toto` using Figure 4.1 as an example. The project owner Diana will create a layout that describes three steps and three functionaries for each step. The first step, tag, will produce a file foo.c to be input into the build step, as well as a foo.po localization file. The second step, build, will use the foo.c file from the tag step and produce a foo binary. Finally, the package step will take the foo.po and foo files and produce a package installable by the end user.

For a more complete and thorough description of all the fields, signature schemes, implementations, a layout editing tool and more, refer to the resources on the project website: `https://in-toto.io`.

4.4.1 The Supply Chain Layout

The supply chain layout explicitly defines the expected layout of the software supply chain. This way, end users can ensure that its integrity is not violated upon verification. To do this, the layout contains the following fields:

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "<README>",
  "keys" : {"<KEYID>":"<PUBKEY_OBJECT>"...},
  "steps" : ["<STEP>", "..." ],
  "inspections" : ["<INSPECTION>","..." ]
}
```

Listing 4.1: The supply chain layout structure

The overarching architecture of the layout definition includes the following relevant fields:

- An expiration date: this will ensure that the supply chain information is still fresh, and that old delivered products can not be replayed to users.
- A readme field: this is intended to provide a human-readable description of the supply chain.
- A list of public keys: these keys belong to each functionary in the supply chain and will be assigned to different steps to ensure that only the right functionary performs a particular step in the supply chain.
- A list of steps: these are the steps to be performed in the supply chain and by who. Step definitions, described in depth in Section 4.4.1, will contain a series of requirements that limit the types of changes that can be done in the pipeline and what functionary can sign link metadata to attest for its existence.
- A list of inspections: these are the inspections to be performed in the supply chain. As described in depth in section 4.4.1, inspections are verification steps to be performed on the delivered product by the client to further probe into its completeness and accuracy.

Though its structure is quite simple, the layout actually provides a detailed description of the supply chain topology. It characterizes each of the steps, and defines any possible requirements for every step. Likewise, it contains instructions for local inspection routines (e.g., verify that every file in a tar archive was created by the right party in the supply chain), which further ensure the delivered product has not been tampered with. As such the layout allows the project owner to construct the necessary framework for a secure supply chain.

For our example supply chain, Diana would have to list the public keys as described on Listing 4.2, as well as all the steps.

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "foo.pkg supply chain",
  "keys" : {"<BOBS_KEYID>":"<PUBKEY>",
            "<ALICES_KEYID":"<PUBKEY>",
            "<CLARAS_KEYID":"<PUBKEY>"},
  "steps" : [{"name": "tag", "..."},
             {"name": "build", "..."},
             {"name": "package", "..."} ],
  "inspections" : ["{"name": "inspect", "..."}]
}
```

Listing 4.2: The supply chain for our example

As described, the layout file already limits all actions to trusted parties (by means of their public keys), defines the steps that are carried out (to limit the scope of any step) and specifies verification routines that are used to dive into the specifics of a particular supply chain. We will describe the latter two fields in depth now.

Step definition

```
{ "_name": "<NAME>",
  "threshold": "<THRESHOLD>",
  "expected_materials": [["<ARTIFACT_RULE>"], "..."],
  "expected_products": [["<ARTIFACT_RULE>"], "..."],
  "pubkeys": ["<KEYID>", "..." ],
  "expected_command": "<COMMAND>"
}
```

Listing 4.3: A supply chain step in the supply chain layout

Every step of the supply chain contains the following fields:

- **name**: A unique identifier that describes a step. This identifier will be used to match this definition with the corresponding pieces of link metadata.
- **expected_materials**: The materials expected as input `ARTIFACT_RULES` as described in Section 4.4.1. It serves as a master reference for all the artifacts used in a step.
- **expected_products**: Given the step's output information, or *evidence*, what should be expected from it? The expected products also contains a list of `ARTIFACT_RULES` as described in section 4.4.1.

- **expected_command**: The command to execute and any flags that may be passed to it.
- **threshold**: The minimum number of pieces of signed link metadata that must be provided to verify this step. This field is intended for steps that require a higher degree of trust, so multiple functionaries must perform the operation and report the same results. For example, if the threshold is set to k, then at least k pieces of signed link metadata need to be present during verification.
- **a list of public keys id's**: The id's of the keys that can be used to sign the link metadata for this step.

The fields within this definition list will indicate requirements for the step identified with that name. To verify these requirements, these fields will be matched against the link metadata associated with the step. The `expected_materials` and `expected_products` fields will be used to compare against the materials and products reported in the link metadata. This ensures that no disallowed artifacts are included, that no required artifacts are missing, and the artifacts used are from allowed steps who created them as products. Listing 4.4 contains the step definition for the build step for our example Layout above.

```
{ "_name": "build",
  "threshold": "1",
  "expected_materials": [
    ["MATCH", "foo.c", "WITH",
     "PRODUCTS", "FROM", "tag"]
  ],
  "expected_products": [["CREATE", "foo"]],
  "pubkeys": ["<BOBS_PUBKEY>"],
  "expected_command": "gcc foo.c -o foo"
}
```

Listing 4.4: The build step in our example layout

Inspection definition

Inspection definitions are nearly identical to step definitions. However, since an inspection causes the verifier on the client device to run a command (which can also generate artifacts), there cannot be a threshold of actions. The other fields are identical to the link metadata generated by a step.

Artifact rules

Artifact rules are central to describing the topology of the supply chain by means of its artifacts. These rules behave like firewall rules and describe whether an artifact should be consumed down the chain, or if an artifact can be created or modified at a specific step. As such, they serve two primary roles: to limit the types of artifacts that a step can create and consume; and to describe the flow of artifacts between steps.

For the former, a series of rules describes the operation within the step. A rule, such as `CREATE`, indicates that a material must not exist before the step is carried out and must be reported as a product. Other rules, such as `MODIFY`, `DELETE`, `ALLOW` and `DISALLOW` are used to further limit what a step can register as artifacts within the supply chain. These rules are described in Grammar 4.5 (full definition in Appendix B). An example of a simple `CREATE` rule can be seen on the step definition in Listing 4.4.

```
[CREATE|DELETE|MODIFY|ALLOW|DISALLOW] artifact_pattern
```

Grammar 4.5: Grammar for operations within a step. artifact_pattern is a regular expression for the paths to artifacts.

For the latter, the `MATCH` rule is used by project owners to describe the flow of artifacts *between steps*. With it, a project owner can mandate that, e.g., a buildfarm must only use the sources that were created during a tag-release step or that only the right localization files are included during a localization step. Compared to the rules above, the `MATCH` rule has a richer syntax, as it needs to account for artifacts relocated during steps (e.g,. a packaging step moving .py files to `/usr/lib/pythonX.X/site-packages/` or a build step moving artifacts to a build directory) using the `IN` clause. Grammar 4.6 describes this rule and the Match function describes the algorithm for processing it during verification. An example of a `MATCH` rule used to match the `foo.c` source from tag into the build step is shown in Listing 4.4.

```
MATCH source_pattern [IN prefix]
  WITH <MATERIALS|PRODUCTS> [IN prefix] FROM step_name
```

Grammar 4.6: The match rule grammar. The IN clauses are optional and source_pattern is a regular expression

4.4.2 Link Metadata Files

Link metadata serves as a record that the steps prescribed in the layout actually took place. Its fields show *evidence* that is used for verification by the client. For example, the *materials* and *products* fields of the metadata are used to ensure that no intermediate products were altered in transit before being used in a step.

In order to determine if the executed step complies with its corresponding metadata, several types of information need to be gathered as evidence. A link includes the following fields:

```
{ "_type" : "link",
```

function MATCH
Input: source_artifacts; destination_artifacts, rule
Output: *result: (SUCCESS/FAIL)*

```
1: // Filter source and destination materials using the rule's patterns
2: source_artifacts_filtered = filter(rule.source_prefix + rule.source_pattern, source_artifacts)
3: destination_artifacts_filtered = filter(rule.destination_prefix + rule.destination_pattern, destination_artifacts)
4: // Apply the IN clauses, to the paths, if any
5: for artifact in source_artifacts_filtered do
6:     artifact.path -= rule.source_in_clause
7: end for
8: for artifact in destination_artifacts_filtered do
9:     artifact.path -= rule.destination_in_clause
10: end for
11: // compare both sets
12: for artifact in source_artifacts_filtered do
13:     destination_artifact = find_artifact_by_path(destination_artifacts, artifact.path)
14:     // the artifact with this path does not exist?
15:     if destination_artifact == NULL then
16:         return FAIL
17:     end if
18:     // are the files not the same?
19:     if destination_artifact.hash != artifact.hash then
20:         return FAIL
21:     end if
22: end for
23: // all of the files filtered by the source materials exist
24: return SUCCESS
```

```
"_name" :  "<NAME>",
"command" : "<COMMAND>",
"materials": {"<PATH>": "<HASH>", "..." : "..." },
"products": {"<PATH>": "<HASH>", "..." : "..." },
"byproducts": { "stdin": "", "stdout": "",
        "return-value": "" },
"environment": {"variables": "<ENV>",
        "filesystem": "<FS>",  ...}
}
```

Listing 4.7: Link metadata format

- Name: This will be used to identify the step and to match it with its corresponding definition inside the layout.
- Material(s): Input file(s) that were used in this step, along with their cryptographic hashes to verify their integrity.
- Command: The command run, along with its arguments.

- Product(s): The output(s) produced and its corresponding cryptographic hash.
- Byproduct(s): Reported information about the step. Elements like the standard error buffer and standard output buffer will be used.
- Signature: A cryptographic signature over the metadata.

Of these fields, the `name`, `materials`, and `products` fields are the counterpart of the fields within the layout definition. This, along with a cryptographic signature used to authenticate the functionary who carried out the step can be used to provide a baseline verification of the supply chain topology (i.e., the steps performed and how do they interrelate via their materials and products). For example, the build step metadata described in Listing 4.8 shows the file `foo.c` used as a material and the product `foo` as created in the build step.

The *byproducts* field is used to include other meaningful information about a step's execution to further introspect into the specifics of the step that was carried out. Common fields included as byproducts are the standard output, standard error buffers and a return value. For example, if a command exited with non-zero status, then the byproduct field be populated with a value such as `return-value: "126"`. In this case, inspections can be set up to ensure that the return value of this specific command must be 0.

```
{ "_type": "link",
  "name": "build",
  "command": ["gcc", "foo.c", "-o", "foo" ],
  "materials": {"foo.c": { "sha256": "bff95e..."}},
  "products": {"foo": {"sha256": "25c696..."}}
  "byproducts": { "return-value": 0,
         "stderr": "", "stdout": ""},
  "environment": {},
}
```

Listing 4.8: The link for the build step

Having a software supply chain layout along with the matching pieces of link metadata and the delivered product are all the parts needed to perform verification. We will describe verification next.

4.4.3 Verifying the Delivered Product

Verification occurs when the link metadata and the layout are received by the client and upon installing the delivered product. A standalone or operating-system tool will perform the verification, as described in the function Verify_Final_Product. To do this, the user will need an initial public key that corresponds to the supply chain layout, as distributed by a trusted channel or as part of the operating system's installation [193].

The end user starts the verification by ensuring that the supply chain layout provided was signed with a trusted key (lines 2-3) and by checking the layout expiration date to make sure the layout is still valid (lines 5-6). If these checks pass, the public keys of all the functionaries are loaded from the layout (line 8). With the keys loaded, the verification routine will start iterating over all the steps defined in the layout and make sure there are enough pieces of link metadata signed by the right functionaries to match the threshold specified for that role (lines 10-20). If enough pieces of link metadata could be loaded for each of the steps and their signatures pass verification, then the verification routine will apply the artifact rules and build a graph of the supply chain using the artifacts recorded in the link metadata (lines 22-23). If no extraneous artifacts were found and all the `MATCH` rules pass, then inspections will be executed[1] (line 25-26). Finally, once all inspections were executed successfully, artifact rules are re-applied to the resulting graph to check that rules on inspection steps match after inspections are executed, because inspections may produce new artifacts or re-create existing artifacts (lines 28-29). If all verifications pass, the function will return `SUCCESS`.

With this verification in place, the user is sure that the integrity of the supply chain is not violated, and that all requirements made by the project's maintainers were met.

4.4.4 Layout and Key Management

A layout can be revoked in one of two ways, the choice being up to the project owner. One is by revoking the key that was used to sign the layout, the other is by superseding/updating the layout with a newer one. To update a layout, the project owner needs to replace an existing layout with a newer layout. This can be used to deal with situations when a public key of a misbehaving functionary needs to be changed/revoked, because when the project owner publishes a newer layout without that public key, any metadata from such misbehaving functionary is automatically revoked. Updating a layout can also be used to address an improperly designed initial layout. The right expiration date for a layout depends on the operational security practices of the integrator.

4.5 Security Analysis

`in-toto` was designed to protect the software supply chain as a whole by leveraging existing security mechanisms, ensuring that they are properly set up and relaying this information to a client upon verification. This allows the client to make sure that all the operations were properly performed and that no malicious actors tampered with the delivered product.

[1]Inspections are executed only after all the steps are verified to avoid executing an inspection on an artifact that a functionary did not create.

To analyze the security properties of in-toto, we need to revisit the goals described in Section 4.2. Of these, the relevant goals to consider are *supply chain layout integrity*, *artifact flow integrity*, and *step authentication*. In this section, we explore how these properties hold, and how during partial key compromise the security properties of in-toto degrade gracefully.

in-toto's security properties are strictly dependent on an attacker's level of access to a threshold of signing keys for any role. These security properties degrade depending on the type of key compromise and the supply chain configuration.

4.5.1 Security Properties With no Key Compromise

When an attacker is able to compromise infrastructure or communication channels but not functionary keys, in-toto's security properties ensure that the integrity of the supply chain is not violated. Considering our threat model in Section 4.2, and contrasting it to in-toto's design which stipulates that the supply chain layout and link metadata are signed, we can assert the following:

- An attacker cannot interpose between two consecutive steps of the supply chain because, during verification, the hash on the products field of the link for the first step will not match the hash on the materials field of the link for the subsequent step. Further, a completely counterfeit version of the delivered product will fail validation because its hash will not match the one contained in the corresponding link metadata. Thus, **artifact flow integrity holds**.
- An attacker cannot provide a product that is missing steps or has its steps reordered because the corresponding links will be missing or will not be in the correct order. The user knows exactly which steps and in what order they need to be performed to receive the delivered product. As such, **supply chain layout integrity holds**.
- Finally, an attacker cannot provide link metadata for which he does not have permission to provide (i.e., their key is not listed as one that can sign link metadata for a certain step). Thus, **step authentication holds**.

However, it is important to underline that this threat model requires that the developer's host systems are not compromised. Likewise, it assumes that there are no rogue developers wishing to subvert the supply chain. For practical purposes, we consider a rogue functionary to be equivalent to a functionary key compromise. Hence this section frames attacks from the standpoint of a key compromise, even when the issue may be executed as a confused deputy problem or a similar issue with equivalent impact.

4.5.2 Security Properties if There is a Key Compromise

`in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. To further explore the consequences of key compromise, we outline the following types of attacks in the supply chain:

- Fake-check: a malicious party can provide evidence of a step taking place, but that step generates no products (it can still, however, generate byproducts). For example, an attacker could forge the results of a test suite being executed in order to trick other functionaries into releasing a delivered product with failing tests.
- Product Modification: a malicious party is able to provide a tampered artifact in a step to be used as material in subsequent steps. For example, an attacker could take over a buildfarm and create a backdoored binary that will be packaged into the delivered product.
- Unintended Retention: a malicious party does not destroy artifacts that were intended to be destroyed in a step. For example, an attacker that compromises a cleanup step before packaging can retain exploitable libraries that will be shipped along with the delivered product.
- Arbitrary Supply Chain Control: a malicious party is able to provide a tampered or counterfeit delivered product, effectively creating an alternate supply chain.

Functionary Compromise

A compromise on a threshold of keys held for any functionary role will only affect a specific step in the supply chain to which that functionary is assigned to. When this happens, the **artifact flow integrity** and **step authentication** security properties may be violated. In this case, the attacker can arbitrarily forge link metadata that corresponds to that step.

The impact of this may vary depending on the specific link compromised. For example, an attacker can fabricate an attestation for a step that does not produce artifacts (i.e., a fake-check), or create malicious products (i.e., a product modification), or pass along artifacts that should have been deleted (i.e., an unintended retention). When an attacker creates malicious products or fails to remove artifacts, the impact is limited by the usage of such products in subsequent steps of the chain. Table 4.1 describes the impact of these in detail from rows 2 to 5 (row 1 captures the case when the attacker does not compromise enough keys to meet the threshold defined for a step). As a recommended best practice, we assume there is a "`DISALLOW *`" rule at the end of the rule list for each step.

It is of note from Table 4.1 that an attacker who is able to compromise crucial steps (e.g., a build step) will have a greater impact on the client than one which, for example, can only

Keys Compromised	Compromised Step Rule	Subsequent Step Rule	Impact
Under threshold	Regardless of rule	Regardless of rule	None
Step	None	Regardless of rule	Fake-check
Step	`ALLOW pattern1` `DELETE pattern2`	`MATCH pattern*`	Unintended Retention
Step	`[ALLOW | CREATE | MODIFY] pattern`	`MATCH pattern`	Product Modification
Layout	N/A	N/A	Arbitrary Supply Chain Control

Table 4.1: Key compromise and impact based on the layout characteristics.

alter localization files. Further, a compromise in functionary keys that do not create a product is restricted to a fake check attack (row two). To trigger an unintended retention, the first step must also have rules that allow for some artifacts before the `DELETE` rule (e.g., the `ALLOW` rule with a similar artifact pattern). This is because rules behave like artifact rules, and the attacker can leverage the ambiguity of the wildcard patterns to register an artifact that was meant to be deleted. Lastly, note that the effect of product modification and unintended retention is limited by the namespace on such rules (i.e., the `artifact_pattern`).

Mitigating risk. As discussed earlier, the bar can be raised against an attacker if a role is required to have a higher threshold. For example, two parties could be in charge of signing the tag for a release, which would require the attacker to compromise two keys to successfully subvert the step.

Finally, further steps and inspections can be added to the supply chain with the intention of limiting the possible transformations on any step. For example, as shown in Section 4.6, an inspection can be used to dive into a Python's wheel and ensure that only Python sources in the tag release are contained in the package.

Project Owner Compromise

A compromise of a threshold of keys belonging to the project owner role allows the attacker to redefine the layout, and thereby subvert the supply chain completely. However, like with step-level compromises, an increased threshold setting can be used to ensure an attacker needs to compromise many keys at once. Further, given the way `in-toto` is designed, the layout key is designed to be used rarely, and thus it should be kept offline.

4.5.3 User Actions in Response to `in-toto` Failures

Detecting a failure to validate `in-toto` metadata involves making a decision about whether verification succeeded or whether it failed and, if so, why. The user's device and the reason for failure are likely to be paramount in the user's decision about how to respond. If the user is installing in an ephemeral environment on a testing VM, they may choose to ignore the warning

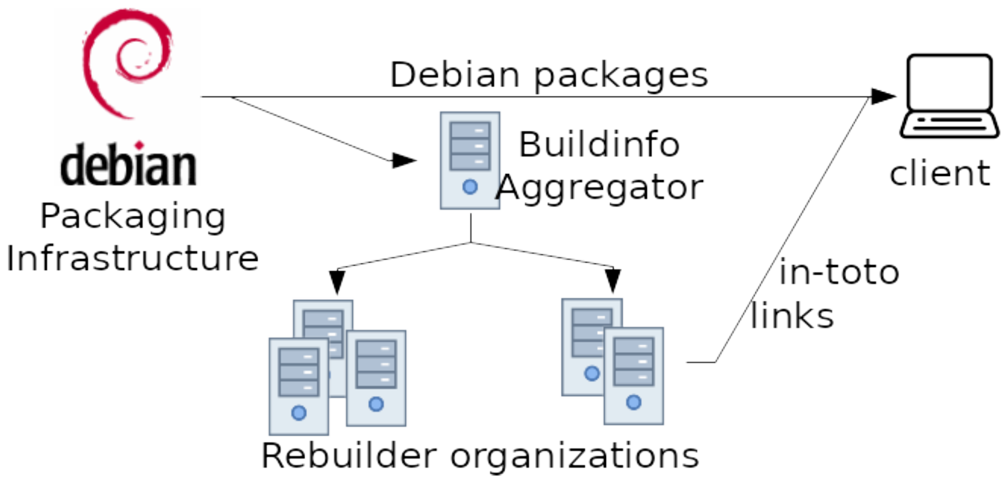


Figure 4.2: The rebuilder setup.

and install the package regardless. If the user is installing in a production environment processing PCI data, the failure to validate in-toto metadata will be a serious concern. So, we expect users of in-toto will respond in much the same way as administrators do today for a package that is not properly signed.

4.6 Deployment

in-toto has about a dozen different integrations that protect software supply chains for millions of end users. This section uses three such integrations to examine how threshold signing, metadata generation, and end-to-end verification function in practical deployments of in-toto.

4.6.1 Debian Rebuilder Constellation

Debian is one of the biggest stakeholders in the reproducible builds project [66], an initiative to ensure bit-by-bit deterministic builds of a source package. One of the main motivations behind reproducible builds is to avoid backdooring compilers [244] or compromised toolchains in the Debian build infrastructure. in-toto helps Debian achieve this goal via its step-thresholding mechanism.

The apt-transport [41] for in-toto verifies the trusted rebuilder metadata upon installing any Debian package. Meanwhile, various institutions (that range from private to non-profit and educational) run rebuilder infrastructure to rebuild Debian packages independently and produce attestations of the resulting builds using in-toto link metadata. This way, it is possible to cryptographically assert that a Debian package has been reproducibly built by a set of *k* out of

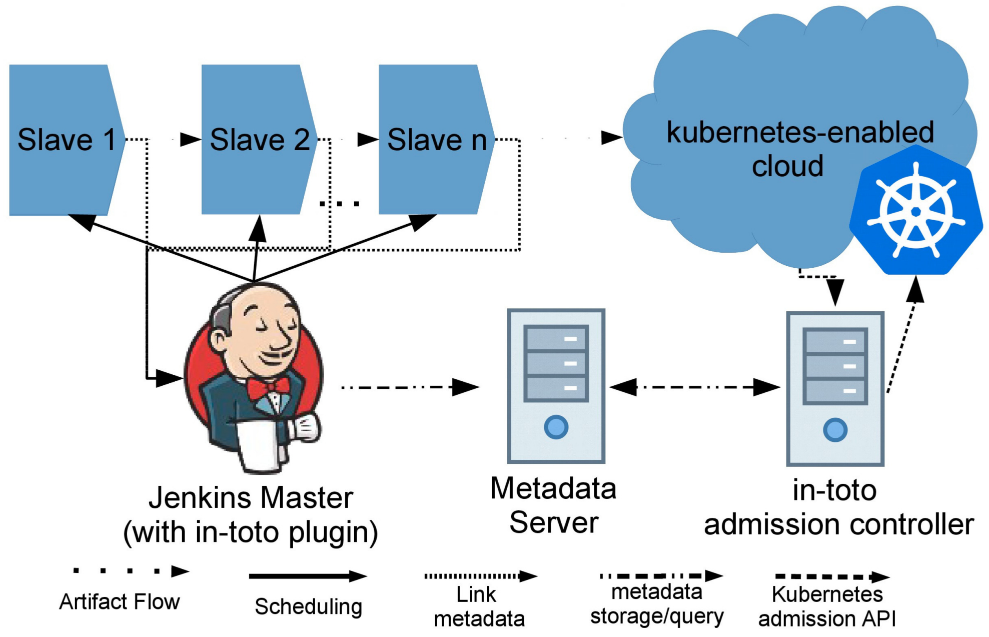


Figure 4.3: The kubesec supply chain.

n rebuilders. Figure 4.2 shows a graphical description of this setup.

By using the `in-toto` verifiable transport, users can make sure that no package was tampered with unless an attacker is also able to compromise at least k rebuilders and the Debian buildfarm. Throughout this deployment, we were able to test the thresholding mechanism, as well as practical ways to bootstrap project owner signatures through the existing package manager trust infrastructure [82, 84].

Deployment insight. Through our interaction with reproducible builds, we were able to better understand how the thresholding mechanism can be used to represent concepts such as a build's reproducibility and how to build `in-toto` into operating-system tools to facilitate adoption.

4.6.2 Cloud Native Builds with Jenkins and Kubernetes

"Cloud native" is used to refer to container-based environments [9]. Cloud native ecosystems are characterized by rapid changes and constant re-deployment of the internal components. They are generally distributed systems, and often managed by container orchestration systems such as Kubernetes [62] or Docker Swarm [20]. Thus, their pipelines are mostly automated using pipeline managers such as Jenkins [44] or Travis [246]. In this type of ecosystems, a host- and infrastructure-agnostic, automated way to collect supply-chain metadata is necessary not only for security, but also for auditing and analyzing the execution of build processes that led to the creation of the delivered product.

In the context of cloud native applications, `in-toto` is used by Control Plane to track the build and quality-assurance steps on kubesec [47], a Kubernetes resource and configuration static analyzer. In order to secure the kubesec supply chain, we developed two `in-toto` components: a Jenkins plugin [32] and a Kubernetes admission controller [21, 42]. These two components allow us to track all operations within a distributed system, both of containers and aggregate `in-toto` link metadata, to verify any container image before it is provisioned. Figure 4.3 shows a (simplified) graphical depiction of their supply chain.

This deployment exemplifies an architecture for the supply chains of cloud native applications, in which new container images, serverless functions and many types of deployments are quickly updated using highly-automated pipelines. In this case, a pipeline serves as a coordinator, scheduling steps to worker nodes that serve as functionaries. These functionaries then submit their metadata to an `in-toto` metadata store. Once a new artifact is ready to be promoted to a cloud environment, a container orchestration system queries an `in-toto` admission controller. This admission controller ensures that every operation on this delivered product has been performed by allowed nodes and that all the artifacts were acted on according to the specification in the `in-toto` layout.

Deployment insight. Our interaction with kubesec forced us to investigate other artifact identifiers such as container images (in addition to files). While `in-toto` can be used today to track container images, the ability to point to an OCIv2 [54] image manifest can provide a more succinct link metadata representation and will be part of future work.

4.6.3 Datadog: E2E Verification of Python Packages

Datadog is a monitoring service for cloud-scale applications, providing monitoring of servers, databases, tools, and services, through a software-as-a-service-based data analytics platform [17]. It supports multiple cloud service providers, including Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, and Red Hat OpenShift. At the time of writing, it has over 8,000 customers, and collects trillions of monitoring record points per day.

The Datadog *agent* is software that runs on hosts. It collects events and metrics from hosts and sends them to Datadog, where customers can analyze their monitoring and performance data. The agent *integrations* are plug-ins that collect metrics from services running on customer infrastructure. Presently, there are more than one hundred integrations that come installed out-of-the-box with the Agent.

Datadog developers wanted an ability to automatically build and publish new or updated integrations independently of agent releases. This is so interested users can try new or updated integrations as they become available, and test whether they are applicable to their needs.

This section will cover how Datadog built the first tamper-evident pipeline using `in-toto`

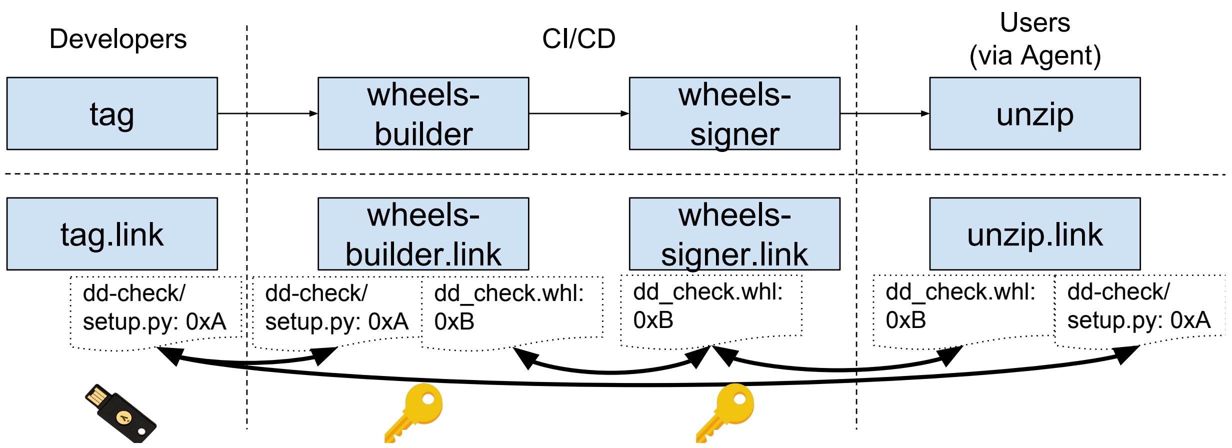


Figure 4.4: The simplified Datadog agent integrations supply chain. There are three steps (`tag` step, `wheels-builder` step, `wheels-signer` step), and one inspection. Arrows denote `MATCH` rules.

and how it leveraged TUF to safely bootstrap key distribution and provide replay-protection and freshness guarantees to in-toto metadata.

End-to-end verification with `in-toto`. The Datadog agent integrations supply chain, shown in 4.4, has three steps:

1. The first `tag` step outputs Python source code as products. Every integration consists of Python source code and several YAML [241] configuration files. The link for this step is signed using a Yubikey hardware dongle [70]
2. In the second `wheels-builder` step, the pipeline must receive the same source code from the `tag` step and produce a Python wheel [63], as well as its updated Python metadata. Each wheel is a ZIP file and its metadata is an HTML file that points to all the available *versions* of an integration.
3. In the third `wheels-signer` step, the pipeline must receive, as materials, the same products as the `wheels-builder` step. This steps signs for all wheels using the system described in the next subsection. It can be dangerous packaging Python source code, because arbitrary code can be executed during the packaging process, which can be inserted by compromising the GitHub repository. Therefore, this step is separate from the `wheels-builder` step, so that a compromise of the former does not yield the signing keys of this step.

Finally, there is one inspection, which first ensures that a given wheel matches the materials of the `wheels-signer` step. It then extracts files from the wheel and checks that they correspond to exactly the same Python source code and YAML configuration files as the products of the `tag` step. Thus, this layout provides end-to-end verification: it prevents a compromised pipeline from causing users to trust wheels with source code that was never released by Datadog developers.

Transport with The Update Framework (TUF). Whereas `in-toto` provides end-to-end verification of the Datadog pipeline, it does not solve a crucial problem that arises in practice: How to securely distribute, revoke, and replace the public keys used to verify the `in-toto` layout. This mechanism must be *compromise-resilient* [181–183,222], and resistant to a compromise of the software repository or server used to serve files. While SSL / TLS protects users from man-in-the-middle (MiTM) attacks, it is not compromise-resilient, because attackers who compromise the repository can simply switch the public keys used to verify `in-toto` layout undetected, and thus defeat end-to-end verification. Likewise, other solutions, such as X509 certificates do not support necessary features such as in-band key revocation and key rotation.

The Update Framework (TUF) [181–183,222] provides precisely this type of compromise-resilient mechanism, as well as in-band key revocation and key rotation. To do so, TUF adds a higher layer of signed metadata to the repository following two design principles that inspired the `in-toto` design. The first is the use of *roles* in a similar fashion to `in-toto`, so that a key compromise does not necessarily affect all *targets* (i.e., any Python wheels, or even `in-toto` metadata). The second principle is minimizing the risk of a key compromise using *offline* keys, or signing keys that are kept off the repository and pipeline in a cold storage mechanism, such as safe deposit boxes, so that attackers who compromise the infrastructure are unable to find these keys.

TUF is used within the Datadog integrations downloader to distribute, in a compromise-resilient manner, the: (1) root of trust for all wheels, TUF and `in-toto` metadata, (2) `in-toto` layout, and (3) public keys used to verify this layout. TUF also guarantees that MiTM attackers cannot tamper with the consistency, authenticity, and integrity of these files, nor rollback or indefinitely replay `in-toto` metadata. This security model is simplified because it ignores some considerations that are out of the scope of this thesis.

In summary, the Datadog pipeline uses TUF to appropriately bootstrap the root of the trust for the entire system, and `in-toto` to guarantee that the pipeline packaged exactly the source code signed by one of the Datadog developers inside universal Python wheels. By tightly integrating TUF and `in-toto`, DataDog's users obtain the compromise resilience of both systems combined.

Deployment insight. Through the Datadog deployment, we learned how to use other last-mile systems like TUF to provide not only compromise-resilience, but also replay-protection, freshness guarantees, and mix-and-match protection for in-toto metadata.

4.7 Evaluation

We evaluated `in-toto`'s ability to guarantee software supply chain integrity on two fronts: efficiency and security. We set off to answer the following questions:

- Does `in-toto` incur reasonable overheads in terms of bandwidth, storage overhead and verification time?
- Can `in-toto` be used to protect systems against real-life supply chain compromises?

In order to answer the first question, we explored `in-toto` as used in the context of Datadog for two reasons: Datadog offers more than 111 integration packages to verify with `in-toto`, and its data and source code is publicly available. Furthermore, it is a production-ready integration that can be used by DataDog's more than 8,000 clients today [81]. Their clients include major companies like Twitter, NASDAQ and The Washington Post [11].

Then, we surveyed historical supply chain compromises and cataloged them. We evaluated these compromises against the `in-toto` deployments described in Section 4.6, accounting for their supply chain configuration, and including the actors involved and their possible key assignments. By studying the nature of each compromise, we were able to estimate what degree of key compromise could hypothetically happen and, with it, the consequences of such a compromise on these supply chains when `in-toto` is in place.

4.7.1 `in-toto`'s Overhead in the Datadog Deployment

In the three major costs that `in-toto` presents are the storage, transfer and verification cost. In order to explore these costs, we set out to use the publicly available Datadog agent integration client and software repository. From this data, we can derive the cost of storing `in-toto` metadata in the repository, the cost of transferring the `in-toto` metadata for any given package and the verification times when installing any package.

Storage overhead. In order to understand the storage overhead, we mirrored the existing agent integrations Python package repository. Then, we inspected the package payloads and the repository metadata to show the cost breakdown of the repository as a whole. Table 4.2 depicts the cost breakdown of the Datadog repository, as mirrored on February 8 of 2019.

Table 4.2 shows that `in-toto` takes up about 19% of the total repository size, and thus makes it a feasible solution in terms of storage overhead. In addition, compared to its co-located security system TUF, the cost of using `in-toto` is higher, with almost four times the metadata storage cost. Further, the breakdown also indicates that the governing factor for this storage overhead are the `in-toto` links, rather than the layout file, with a layout being approximately 6 to 3 times smaller than the links (42 KB in comparison of the 148KB for all the link metadata).

Type	total package	Python metadata	TUF	in-toto links	in-toto layout
RSA 4096	74.02%	0.83%	5.51%	16.75%	2.89%
DSA 1024 & ed25519	74.48%	0.84%	5.54%	16.35%	2.79%
optimized	79.56%	0.90%	5.92%	10.65%	2.97%

Table 4.2: Storage overhead breakdown for a in-toto enabled package repository. All metadata is compressed using zlib.

There are two main reasons that drive this cost. First and foremost, is the engineering decision to track all the files within the pipeline (including Python metadata). Although these are not required to be tracked with in-toto, for the sake of security (as this type of metadata is being protected by TUF), it eases the implementation at a manageable cost. The second is that of signatures: the signatures used within the Datadog deployment come from either 4096-bit OpenPGP keys on a Yubikey, or 4096-bit PEM keys. These alone account for almost half of the in-toto metadata size.

For this reason, it is possible to further reduce the size of the metadata to 13% of the total repository size. Rows two and three of Table 4.2 represent the repository overhead when limiting the amount of files tracked to only Python sources and packages and using a DSA1024 as the signing algorithm.

Network overhead. Similar to storage overhead, the network overhead incurred by clients when installing any integration is of utmost importance. To explore this cost, we investigate the raw package sizes and contrast it with the size of the package-specific metadata size. It is of note though, that the size of in-toto metadata does not scale with the size of the package, but rather the number of files inside of it. This is because most of the metadata cost is taken by pieces of link metadata, of which the biggest three fields are the signature, expected_materials and expected_products. Figure 4.5 shows both the distribution of package sizes and the distribution of metadata sizes in increasing order.

In Figure 4.5 we can see that, for most packages, the metadata size cost is below 44% of the package size. In fact for the 90th percentile, the metadata cost approaches a costly 64%, to a worst case of 103%. However, upon inspecting these cases, we found that the issue is that these are cases in which *link metadata is tracking files not contained in the delivered product*. Indeed, in-toto is tracking files, such as test suites, fixtures and even iconography that does not get packaged on the integrations Python wheel. The worst case scenario is in fact cisco_aci, which only packages 12 files out of 316 contained in the tag step metadata.

Verification overhead. Finally, to draw insight from the computation time required to verify each package, we ran a series of micro-benchmarks on a laptop with an Intel i7-6500U processor and 8GB of RAM. In this case, we ran an iterated verification routine with the packages already fetched and instrumented the Datadog agent installer to measure the installation time with and

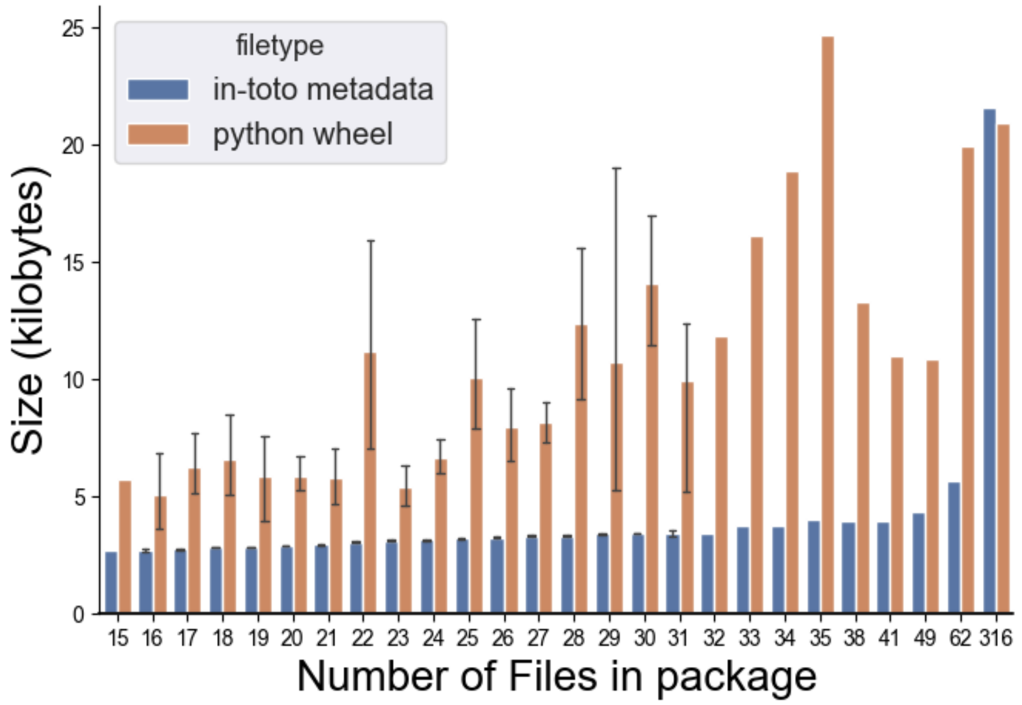


Figure 4.5: Package and metadata size distribution. Error bars show packages with the same number of files but different sizes.

without `in-toto` verification.

From this experiment, we conclude that `in-toto` verification adds less than 0.6 seconds on all cases. This is mostly dominated by the signature verification, and is thus bounded by the number of links to verify (i.e., the number of steps times the threshold).

4.7.2 Supply Chain Data Breaches

We surveyed 30 major different supply chain breaches and incidents occurring from January 2010 to January 2019 (this list of historical attacks is included in Appendix D). These historical incidents cover a variety of software products and platforms, such as Apple's Xcode [207], Android GTK [22], MeDoc financial software [86], Adobe updater [175], PHP PEAR repository [83], and South Korean organizations [247].

Studying these historical attacks identified the type of access that the attacker had (or was speculated to have) and identified three categories: the attacker had control of infrastructure (but not functionary keys), the attacker had control over part of the infrastructure or keys of a specific functionary, and the attacker was able to control the entire supply chain by compromising a project owner's infrastructure (including their signing key).

For the historical attacks in Appendix D, we determined whether an attack used a compromised key, and then labeled those attacks with "Key Compromise". We also determined the

Attack Name	Datadog	Reproducible Builds	Cloud-Native
Keydnap [142]	✓	✓	✓
backdoored-pypi [104]	✓	✓	✓
CCleaner Atatck [230]	✓	✓	✗
RedHat breach [264]	✓	✓	✗
*NotPetya [86]	✓	✗	✗
Operation Red [247]	✓	✗	✗
KingSlayer [218]	✓	✗	✗

Table 4.3: The impact of the historical attacks on the three `in-toto` deployments. A ✓indicates that the deployment could have detected the attack.

degree of access in the attack (all the way to the possible step) and labeled each attack with an "Access Level" that indicates the step in the chain where the attack took place.

We now analyze how these compromises could affect the three supply chains where `in-toto` was deployed (as described in Section 4.6). Our analysis indicates that the majority of attacks (23 out of 30) took place without any key compromise. In those cases, none of the three `in-toto` deployments would have been affected since the client inspection could detect extraneous artifacts or malicious delivered products.

Out of the 30 studied incidents, 7 involved a key compromise. We summarize our analysis of these attacks in Table 4.3. One attack, Keydnap [142], used a stolen Apple developer certificate to sign the malicious software package. Therefore, this attack would not have affected any `in-toto` deployments, because `in-toto` would detect that an unauthorized functionary signed the link metadata. Another attack used the developer's ssh key to upload a malicious Python package on PyPI [104]. All `in-toto` deployments could have detected this attack since files extracted from the malicious package would not exactly match the source code as the products of the first step in the supply chain.

The remaining five attacks involving a key compromise were recent sophisticated incidents that affected many clients and companies. The CCleaner [230] and RedHat [264] attacks are not effective against the Reproducible Builds deployment (RB) and Datadog (DD), as the former implements a threshold mechanism in the build step and the latter does not build binaries in their infrastructure. In a similar flavor, three attacks (Operation Red [247], NotPetya [86], and KingSlayer [218]) would not affect the Datadog deployment, as it implements a threshold mechanism in the packaging step. The Cloud Native deployment, on the other hand, would detect none of these five attacks, as it does not employ thresholds. To conclude, the `in-toto` deployments would detect most of the historical attacks based on the general `in-toto` design principles. For those attacks that involve key compromises, our analysis shows that `in-toto`'s use of thresholds is an effective mechanism.

Key Takeaway. Cloud Native (83%) and reproducible builds (90%) integrations of `in-`

toto would prevent most historical supply chain attacks. However, integration into a secure update system as was done by Datadog (100%) provides further protection.

4.8 Conclusions and future work

In this chapter, we have described many aspects of in-toto, including its security properties, workflow and metadata. We also explored and described several extensions and implications of using in-toto in a number of real-world applications. With this we have shown that protecting the entirety of the supply chain is possible, and that it can be done automatically by in-toto. Further, we showed that, in a number of practical applications, in-toto is a practical solution to many contemporary supply chain compromises.

Although plenty of work needs to be done in the context of the in-toto framework (e.g., decreasing its storage cost), tackling the first major limitations of supply chain security will increase the quality of software products. We expect that, through continued interaction with the industry and elaborating on the framework, we can provide strong security guarantees for future software users.

function VERIFY_FINAL_PRODUCT
Input: layout; links; project_owner_key
Output: *result: (SUCCESS/FAIL)*

```
// verify that the supply chain layout was properly signed
if not verify_signature(layout, project_owner_key) then
    return FAIL
end if
// Check that the layout has not expired
if layout.expiration ¡ TODAY then
    return FAIL
end if
// Load the functionary public keys from the layout
functionary_pubkeys = layout.keys
// verify link metadata
for step in layout.steps do
    // Obtain the functionary keys relevant to this step and its corresponding metadata
    step_links = get_links_for_step(step, links)
    step_keys = get_keys_for_step(step, functionary_pubkeys)
    // Remove all links with invalid signatures
    for link in step_links do
        if not verify_signature(link, step_keys) then
            step_links.remove(link)
        end if
    end for
    // Check there are enough properly-signed links to meet the threshold
    if length(step_links) ¡ step.threshold then
        return error(”Link metadata is missing!”)
    end if
end for
// Apply artifact rules between all steps
if apply_artifact_rules(steps, links) == FAIL then
    return FAIL
end if
// Execute inspections
for inspection in layout.inspections do
    inspections.add(Run(inspection))
end for
// Verify inspections
if apply_artifact_rules(steps + inspections, links) == FAIL then
    return FAIL
end if
return SUCCESS
```

Chapter 5

Related Work

5.1 Software Supply Chains and Their Threats

Identifying software supply chain security threats has been studied for more than 30 years. However, the work that sought to understand their threats has dated around the early 2000's. Likewise, solutions that holistically address software supply chain security will not appear until a decade later.

Identifying common software supply chain practices Pilatti [210] conducted a longitudinal study of software configuration deployments over different ecosystems, geographical locations and industries to derive challenges and best practices for software pipelines. This work tried to identify trends and tendencies in software engineering, and thus did not explore the security practices of the supply chain, or of individual steps in the chain.

Lipner [186] laid the foundation for what would later become Microsoft's Secure Development Lifecycle (SDL), a common set of practices for the development of secure software. Of mention, Lipner does not mention how to enforce these practices are followed, but rather leaves it to organizations to identify and apply the ones that better suit their goals. Both Lipner's and Pilatti's work served as a guides to identify and delineate common software supply chain practices.

Evaluating and mitigating risks A foundation work assessing software supply chain compromises was done by Woody et. al [141], who tried to understand the different types of software supply chain compromises and their mitigations. This work mostly focused on delineating the attack surface (e.g., third party dependencies) of current supply chain practices and develop a maturity level for high-assurance software supply chain scenarios. Woody's work was fundamental for early stages of this work to identify a threat model for the software supply chain.

Work by Kowalski [221] in this area attempted to not only identify the nature of the risks, but provide a framework from which system administrators and security professionals could assess risks in their software pipelines. The goal of this work was to allow users to prioritize and triage practices that will yield a more secure supply chain. It is our belief that Kowaski's work can be well suited to drive policies that can be encoded in an in-toto Layout and may be even seminal pieces for future work on automating best practice policies for the software supply chain.

Plummer et. Al's [225] work relied building automated tools to traverse dependency graphs and find vulnerable components in a software artifact. Of mention, this work is a big component on IonChannel's security offering, and is an example of widespread vulnerability scanners used in industry. Similar work in this vein is that of Duan et Al. which focuses on identifying licensing violations, as well as 1-day security vulnerabilities on Open Source Software components of [136, 138] In contrast, in-toto is focused on ensuring these quality assurance practices are in place and correctly followed.

Software supply chain security. In addition, many software engineering practices have been introduced to increase the security of the software development lifecycle [99,191,192,205,216]. Additional work by Devanbu et al. [134] has explored different techniques to "construct safe software that inspires trust in hosts." These techniques are similar to `in-toto` in that they suggest releasing supply chain information to the end users for verification.

Though none of these proposals suggest an automated tool to ensure the integrity of the supply chain, they do serve as a helpful first step in designing `in-toto`. As such, their practices could be used as templates for safe supply chain layouts.

Finally, there have been hints by the industry to support features that could be provided by `in-toto` [214, 236, 261]. This includes providing certificates noting the presence of a process within the supply chain and providing software transparency through mechanisms such as reproducible builds.

5.2 Securing Software Supply Chains

To the best of our knowledge, work that attempts to use an automated tool to secure the supply chain is scarce. However, there has been a general push to increase the security of different aspects within the supply chain, as well as to tighten the binding between neighboring processes within that chain. In this section, we mention work relevant to supply chain security, as some of it is crucial for the success of `in-toto` as a framework. We also list work that can further increase the security guarantees offered by `in-toto`.

5.2.1 VCS Security

The source code repository is usually the first link in the supply chain. Early work in this field has explored the different security properties that must be included in software configuration management tools [129]. Version control systems, such as Git, incorporate protection mechanisms to ensure the integrity of the source code repository, which include commit hash chaining and signed commits [153, 155].

Wheeler [256] provides an overview of security issues related to software configuration management (SCM) tools. He puts forth a set of security requirements, presents several threat models (including malicious developers and compromised repositories), and enumerates solutions to address these threats. Gerwitz [153] provides a detailed description of creating and verifying Git signed commits. Signing commits allows the user to detect modifications of committed data. Git incorporates protection mechanisms, such as commit signing and commit hash chaining. Unfortunately, they do not prevent the attacks we introduce in this work.

There have been proposals to protect sensitive data from hostile servers by incorporating secrecy into both centralized and distributed version control systems [4, 209]. Shirey et al. [226] analyzes the performance trade-offs of two open source Git encryption implementations. Secrecy from the server might be desirable in certain scenarios, but it is orthogonal to our goals in this work.

The "push certificate" mechanism, introduced in version 2.2.0 of Git, allows a user to digitally sign the reference that points to a pushed object. However, push certificates do not protect against most of the attacks we describe in this work. Furthermore, push certificates were designed for out-of-band auditing (*i.e.* they are not integrated into the usual workflow of Git and need to be fetched and verified by a trusted third party using out-of-band mechanisms). As a result, push certificates are rarely used in practice.

Fork Consistency

A problem that could arise in remote storage used for collaborative purposes is when the untrusted storage server hides updates performed by one group of users from another. In other words, the server equivocates and presents different views of the history of operations to different groups of users. The *fork consistency* property seeks to address this attack by forcing a server that has forked two groups in this way to continue this deception. Otherwise, the attack will be detected as soon as one group sees an operation performed by the other group after the moment the fork occurred.

SUNDR [184] provides fork consistency for a network file system that stores data on untrusted servers. In SUNDR, users sign statements about the complete state of all the files and exchange these statements through the untrusted server. SPORC [144] is a framework for building collaborative applications with untrusted servers that achieves fork* consistency (*i.e.*, a weaker variant of fork consistency). Our solution seeks to achieve a similar property and shares similarities with SUNDR in that Git users leverage the actual Git repository to create and share signed statements about the state of the repository. However, the intricacies and usage model of a VCS system like Git impose a different set of constraints.

Other work, such as Depot [188], focuses on recovering from forks in an automatic fashion (i.e., not only detecting forks, but also repairing after they are detected). Our focus is on detecting the metadata manipulation attacks, after which the affected users can perform a manual rollback procedure to a safe point.

Caelus [178] seeks to provide the same declared history of operations to all clients of a distributed key-value cloud store. Caelus assumes that no external communication channel exists between clients, and requires them to periodically attest to the order and timing of operations by writing a signed statement to the cloud every few seconds. The attestation schedule must be pre-defined and must be known to all clients. Our setting is different, since Git developers usually communicate through multiple channels; moreover, a typical team of Git developers cannot be expected to conform to such an attestation policy in practice. Caelus considers a setting in which clients of the storage service are devices that are all under a user's control (e.g., a user's laptop, tablet, or smartphone).

5.2.2 Buildsystem and Development Security

The field of automated testing and continuous integration has also received attention from researchers. Recently, self-hosted and public automated testing and continuous integration systems have become popular [118,146,246]. Work by Gruhn et al. [163] has explored the security implications of malicious code running on CI systems, showing that it is possible for attackers to affect other projects being tested in the same server, or the server itself. This work, and others [139] serve as a motivation for `in-toto`'s threat model.

Further work by Hanawa et al. [166] explores different techniques for automated testing in distributed systems. The work is similar to `in-toto` in that it allocates hosts in the cloud to automatically run tests for different environments and platforms. However, `in-toto` requires such systems to provide certification (in the form of link metadata) that the tests were run and the system was successful.

Subverting the development environment, including subverting the compiler, can have a serious impact on the software supply chain [243]. Techniques such as Wheeler's diverse double-compiling (DDC) [257] can be used to mitigate such "trusting trust" attacks. In the context of reproducible builds project, DDC can also be used for multi-party verification of compiler executables.

Finally, automated patching of open source applications has been explored widely. Of note, the work by Duan et al. [137] to effectively short-circuit different supply chains for automated Android application patching shows ways to use supply-chain meta information to secure deployed binaries. In this case, in-toto can be used as a messaging mechanism to aid in this forward-patching approach.

5.2.3 Verifying Compilers, Applications and Kernels

Ongoing work on verifying compilers, applications and kernels will provide a robust framework for applications that fully comply with their specification [167,179]. Such work is similar to in-toto in that a specification is provided for the compiler to ensure that their products meet stated requirements. However, in contrast to our work, most of this work is not intended to secure the origin of such specification, or to provide any proof of the compilation's results to steps further down the supply chain. Needless to say, verifying compilers could be part of a supply chain protected with in-toto.

Furthermore, work by Necula et al. introduces proof-carrying code [201, 202], a concept that relies on the compiler to accompany machine code with proof for verification at runtime. Adding to this, industry standards have included machine code signing [94] to be verified at runtime. This work is similar to in-toto in that compilers generate information that will be verified by the end user upon runtime. Although these techniques are more granular than in-toto's (runtime verification vs verification upon installation), they do not aim to secure the totality of the supply chain.

5.2.4 Package Management and Software Distribution Security

Version control systems are often considered a publishing platform [157]. For this reason, security measures have been proposed to version control systems, such as git, to allow them to securely export software artifacts [119]. Although it may appear that these practices are the same as version control system security, the software artifacts hosted in these are generally disjointed and suffer from a lack of linking between the source code checkout (from the VCS) to the released artifact. In this case, in-toto can be used to authenticate the backend process that is automatically generating the release artifacts.

Work by Cappos et al. has been foundational to the design of in-toto's security mechanisms [109,183,222]. The mechanisms used to secure package managers are similar to in-toto in that they rely on key distribution and role separation to provide security guarantees that degrade with partial key compromise. However, unlike in-toto, these systems are restricted to software updates, which limit their scope. Concepts from this line of work could be overlaid on in-toto to provide additional "last mile" guarantees for the resulting product, such as package freshness or protection against dependencies that are not packaged with the delivered product.

5.2.5 Automated Supply Chain Administration Systems

Configuring and automating processes of the supply chain has been widely studied. Works by Bégin et al. [102], Banzai et al., [100] and Andreetto et al. [88] focus on designing supply chains that automatically assign resources and designate parties to take part in different processes to create a product. This work is similar to `in-toto` in that it requires a supply chain topology to carry out the work. However, none of these projects were focused on security. Instead, they deal with adaptability of resources and supply chain automation.

Perhaps most closely related to `in-toto` is the Grafeas API [80] released by Google. However, Grafeas's focus is on tracking and storing supply chain metadata rather than security. Grafeas provides a centralized store for supply chain metadata, which can be later queried by verification tools such as Google's Binary Authorization [161]. Grafeas does not provide a mechanism to describe what steps should be performed, validate performed steps, or even support cryptographic signatures [2]. Finally, `in-toto` is architecture agnostic, while Grafeas is mostly cloud-native; `in-toto` was geared to represent supply chains whether they are cloud-native, off-cloud or hybrid-cloud. At the time of this writing, `in-toto` is now fully supported as a type of Grafeas attestation [1, 2], which allows for the fine-grained scrutiny provided by Grafeas plus the cryptographically-enforced policies provided by in-toto.

5.2.6 Supply Chain Cataloging, Transparency and Software Bills of Materials

There are various initiatives that, although they are not security-focused, may allow for tracking of software artifacts as they flow through chains of custody. These initiatives are focused on providing a platform agnostic and transferable record of software artifacts. In this space, projects like the United States National Telecommunications and Information Administration (NTIA) Software Bill of Materials (SBoM) [52], the Object Management Group (OMG) Software Bill of Materials [71], OpenChain [55], and the Software Package Data Exchange (SPDX) 3.0 [30] specification can be used to provide rich descriptions of various types of software artifacts. These projects are quite useful in that, provided they include integrity and cryptographic authentication for their SBoM's, they can be used in lieu of in-toto link metadata. In fact, the OMG standard and SPDX 3.0 are likely to not only provide these properties but also allow for in-toto links as the generic bill of materials.

Another standard worthy of mention is the the World Wide Web Consortium's (W3C) PROV specification. While this has not enjoyed much deployment, it is an ontology used to identify actors, actions and artifacts of actions within different chains of custody on the web world. It is possible that in-toto can be extended to a fully-compliant PROV implementation to secure web content as it flows through web servers, mirrors, and CDN's.

Finally, there are other whole-chain projects that are attempting to provide transparency into the software supply chain. One that has enjoyed adoption in open source communities is SPARTS [69], which uses attestations similar to link metadata and are hosted in a private blockchain. The difference between SPARTS and in-toto is that in-toto is designed to provide artifact authentication and artifact flow integrity properties, while SPARTS is focused mostly in transparency and compliance.

Within the same transparency family, work by Linderud [185] and Nikitin [203] provides insight on how to use auditable data structures [87] to provide a highly-scalable, tamper-evident log of supply chain steps. As shown by Linderud, these data structures (e.g., a transparency log, or a transparency set) can host in-toto metadata to ensure properties such as build reproducibility or drive insights about artifact lineage. This work proves itself to be a very natural evolution to answer outstanding questions regarding transport and storage mechanisms for in-toto metadata.

Chapter 6

Future Work

6.1 Future in-toto Enhancements

Future projects can leverage other in-toto components. The in-toto official Jenkins [33] plugin already provides enough facilities to provide cryptographically-signed attestations of every single operation on a distributed system managed by any Jenkins master, as well as storing all artifacts using built-in artifact storage. The integration of these artifacts into open standards such as the OCIv2 [54] registry specification will simplify their retrieval, storage and verification.

Given that the software supply chain is a relatively new area, in-toto is a green field for various research thrusts. The focus will be in increasing transparency, strengthening existing software supply chains, and in protecting cyber-physical supply chains:

6.1.1 Supply Chain Transparency and Analysis

in-toto works today by tersely describing the trust relationship between nodes, as well as the flow of artifacts on a distributed pipeline. However, anchoring trust, and the temporal notion of the creation of artifacts is an open problem. By storing in-toto metadata in a temporally-aware data structure, an artifact transparency log, it is possible to scrutinize the evolution of artifacts as they are created. This same technique will also help track artifacts as they become part of larger or broader supply chains.

Finally, we will be able to mine and study supply chain metadata to drive new insights about how is software made. This way, we can create reliable datasets to correlate best practices and the appearance of specific types of vulnerabilities. This work will culminate on creating (possibly third-party approved) best-practice in-toto layouts such that the bar for secure software is raised.

6.1.2 Strengthening Existing Supply Chains

With in-toto, and the insights collected, we will be able to understand the sources of vulnerabilities and, like bugs in software, we will be able to create tools to squash software supply chain bugs. The most prominent example of this is probably the case against toolchain non-determinism — a fundamental problem for security due to the existence of backdooring compilers and compromised toolchains (e.g., XCodeGhost) — which is the main drive behind the reproducible builds project. However, identifying sources of non-determinism on builds has mostly been addressed empirically (i.e., by manually debugging and inspecting a build).

With in-toto, we can look at supply-chain metadata as distributed software execution traces, and use it to replay the creation of software, while selectively varying parts of the toolchain in a node or vary nodes altogether in a similar fashion to how a coverage-based fuzzer works for finding bugs in software. This way we can stress-test reproducibility, find sources of non-determinism during package creation, and raise the assurance of reproduced packages.

6.1.3 Improving Cyber-Physical Supply Chain Security

There is a trove of work on securing the physical supply chain, yet crossing the boundaries between software, hardware and physical systems is an unexplored possibility. There is unprecedented flow from software to produce hardware-accelerated algorithms that would benefit from the cryptographically-signed paper trails of in-toto. The challenge in this angle is to the cross-boundary nature of hardware-accelerated implementations, we will introduce highly-granular taint metadata on in-toto to truthfully attest the source of hardware implementations.

In addition, the other direction: hardware attested software artifacts is something that plenty of users experience everyday. Using in-toto primitives will shed light in IoT firmware development practices and enforce stricter security processes. For this, we will introduce a technique called layered attestation to provide a strong, hardware-level trusted-computing base from which software supply chain execution can take over over a fully-supply-chain-verified hardware layer.

6.2 Fully-Automating the Vulnerability Cataloging Pipeline

A common notion within the security industry is "if it is on the National Vulnerability Database (NVD) it is already late." The reason behind this cynicism is the long delay between a when vulnerability is detected, triaged, confirmed, properly announced, fixed and deployed. While this process is the best we can achieve today, it leaves much to be desired when it comes to impactful vulnerability remediation.

This problem arises because *the vulnerability supply chain* suffers from a lack of automation in crucial steps (e.g., triaging), and verification in others (e.g., cataloging). This imbalance has led to a flood of vulnerability reports by resourceful fuzzing initiatives (e.g., Google's OSSFuzz [57]), leaving project maintainers to manually verify, patch, catalog and publish an advisory. As an example, the GraphicsMagick project — used by most open source software that interacts with images — has 129 publicly announced vulnerabilities via OSSFuzz that have not been patched or even cataloged.

6.2.1 CVE-to-Artifact Provenance

Once a vulnerability is appropriately fixed, there are very few mechanisms that ensure that the fix persists throughout the supply chain. Some CVE's such as CVE-2020-1927 and CVE-2019-10098 [15, 16] note effects on which, after a couple of releases, the same vulnerability resurfaces due to lack of regression testing. Another example of this is CVE-2008-1685, in which the vulnerability surfaced when a compiler optimization removed the a security check (a simple if guard). Some other cases can resurface more than once, like CVE-2015-1283, then turned CVE-2015-2716, then turned CVE-2016-4472, in which compiler optimizations under certain conditions removed overflow checks in the widely used library expat [12–14].

Another, similar case to are CVE's that are confirmed and validated on specific distributions, or with a specific build pipeline in place. When this happens, it is hard for other downstream package providers to verify that this vulnerability is applicable to their configuration.

6.2.2 Non-Reproducible Vulnerabilities

Several false-positives are caused by automated vulnerability exploration software (e.g., fuzzers). In these cases, a false-positive is generated due to a misconception of the tool generating the reproducer (e.g., a non-reachable path due to a guard statement on a coverage-based fuzzer).

Thus, it will be possible to use supply chain semantics to introduce a fully automated CVE registration pipeline using novel techniques such as exploit cross-validation (i.e., using a proof-of-concept to fully verify the exploit pipeline) with dynamically-traced vulnerability descriptions and CVE-to-artifact lineage tracking.

Automating the vulnerability triaging, identification, verification and description process is an attractive, yet almost untapped venue for research. To address these issues the effort is twofold:

6.2.3 Automating CVE Registration and Triaging

Develop the first CNA using automated vulnerability proof-of-concept evaluations. The goal of this is to remove the existing bottleneck in the current state of the CVE supply chain. To do so, we will create a proof-of-concept automatic vulnerability classifier, borrowing ideas from the taint-analysis, provenance community and the sand-boxing community to create descriptive vulnerability definitions and accurate Common Vulnerability Scoring System (CVSS) values. This will require the answer questions regarding meaningful automated vulnerability definitions, triaging, and verification, as well as host sand-boxing and process isolation.

6.2.4 CVE-to-Artifact Lineage

The other question regarding CVE registration is one of granularity. Currently, CVE's are loosely associated with projects, and knowing if a vulnerability appears in different variants of a product (e.g., a Debian package vs the compiled binary on the project's website) is near-impossible to answer. We can use artifact lineage techniques from in-toto to follow different variants of an artifact and automatically verify whether it is vulnerable. This way, we can trace the CVE's target (generally related to source code) and match it to all the artifacts that this source code produced and even other artifacts that in turn were produced from the original artifacts (e.g., statically-linked libraries).

I believe these research areas will change the way we track vulnerabilities today and, hopefully, will finally achieve meaningful and timely vulnerability remediation. However, vulnerability remediation is only impactful if security fixes can be delivered to users in a safe, trustworthy and timely manner. As such, I intend to leverage my existing work on software update system security and fix existing legacy systems dependent on PGP.

6.3 Using in-toto Principles to Provide Practical PGP Crypto-Agility

Today, critical infrastructure, such as operating system images, binary downloads, email among others, use PGP (standardized in RFC4880) for signing and encryption. Unfortunately – as more than 20 years of literature suggest — PGP is quite vulnerable to a myriad of attacks that can subvert its security in many scenarios. Attacks on PGP email signing and encryption, as well as HKP+PGP subpacket spamming have rendered PGP unusable for most applications, and security experts on both academia and industry are pleading users to move away from it once and for all.

However, PGP is embedded into critical infrastructure and processes (e.g., the Debian PGP keyring has thousands of developers and it's the only means to authenticate developers on their geographically distributed, decentralized organization), and those processes are slow-moving and hard to change. For some, it is impossible to even start a dialogue in which PGP is phased out to include other technologies (e.g., TUF). This begs the question, is any part of RFC4880 (and its recent draft RFC4880bis) salvageable? Can we provide a way to continue providing a usable standard with meaningful security guarantees on the current PGP ecosystem?

6.3.1 GPG/PGP Ecosystem Evolution

Gnu Privacy Guard's (or GPG, the most widely-used implementation of PGP) evolution is not as glacial as it appears, and major changes in the default behavior of GPG 1 to GPG 2 have had unintended consequences on different types of applications. Adding to this, GPG also goes beyond RFC4880, and may not fully comply with certain expected behaviors. In this line of research, we will explore how certain elements, such as keyids, long keyids, fingerprints and user id's can be collided, spoofed or replaced to abuse package managers such as Debian's.

6.3.2 ARGO: A Reasonable GPG Overlay

Then, We will propose a pathway for evolution to existing GPG-dependent infrastructure by creating semi-backwards compatible tooling. The principle behind it, will be to track signature objects using in-toto principles in such a way that we can provide transparency of when they are used, when they need to be revoked and where they were used.

This way, existing infrastructure can safely phase out dangerous GPG features first, and then GPG as a whole, while moving onto better-suited ecosystems for their particular use. The first target in mind is to use ARGO to help users migrate package repositories to TUF or a TUF-variant ecosystem while ensuring reasonable security guarantees. Later, I envision tackling issues with identity management, keyservers and key synchronization and finally email encryption/signing.

Having addressed security concerns stemming from the early supply chain, through the vulnerability remediation pipeline, and finally in the last-mile, the update system and end-user applications, I will close the loop by tackling everyday users' (not just developers') security needs. To do this, our focus will move from holistic to an individual-step research project. In this particular part of my work, I will be looking at novel ways to securely store data, primarily passwords, on cloud-native systems and beyond.

6.4 Provenance Mechanisms for Trustworthy Research Artifacts

The area of research artifacts is of special interest. Research artifacts face the challenge of experiment reproducibility in different ecosystems and environments and transparency will bring unprecedented ways to build upon our peer's work, as well as to help settle disputes between contending facts on similar work.

As software production and consumption becomes more structured, software supply chains' quality and security will have a bigger impact on the quality and security of software products themselves. We, computer scientists, are part of a very exciting part of computing history where infrastructure, technology and people's daily lives rely on our work. With this computational-pervasiveness in mind, I am eager to design the new technologies to create a more reliable, secure and robust aspect of everyday life.

Chapter 7

Conclusion

This thesis introduces an holistic view to software supply chain security. To do so, it provided an exploratory analysis of the software supply chain security landscape, followed by a study of security properties of point systems using git as an example. Finally, this thesis introduced in-toto, the first holistic software supply chain security framework.

Although in-toto is currently experiencing wide adoption, it is important to underline that there is plenty to do for securing future supply chains. As these systems become more complex and their practices change, it is very likely in-toto will require extensions to its protocol. In fact, as the time of this writing, work on allowing in-toto to secure abstract entities such as pull-requests or SPDX element identifiers are becoming a reality.

In addition, parallel work by various communities will also serve as an enable to mature the security offering of in-toto. Initiatives such as SPDX 3.0, the various Software Bill of Materials initiatives, and artifact cataloging projects such as Grafeas will allow in-toto to bolster a richer taxonomy of software supply chain events and further enable scrutiny over software supply chain security practices.

in-toto is a need-based project, and as such it will follow closely the evolution of software projects. This way, we can rest assured that future supply chains will be friendly to in-toto's security principles.

Bibliography

[1] Add in-toto: include final corrections, nits. https://github.com/grafeas/grafeas/pull/426.

[2] Add Signature message to v1beta common.proto. #253. https://github.com/grafeas/grafeas/pull/253.

[3] Apache maven. https://maven.apache.org/.

[4] Apso: Secrecy for Version Control Systems. http://aleph0.info/apso/.

[5] Apt. https://wiki.debian.org/Apt.

[6] Archiso. https://wiki.archlinux.org/index.php/Archiso.

[7] Buildbot: Gdb. https://gdb-buildbot.osci.io/#/.

[8] Cargo: the Rust package manager. https://github.com/rust-lang/cargo/.

[9] Cloud native computing foundation. https://www.cncf.io/.

[10] The continuous integration framework. http://buildbot.net/.

[11] Customers — Datadog. https://www.datadoghq.com/customers/.

[12] Cve-2015-1283 detail. https://nvd.nist.gov/vuln/detail/CVE-2015-1283.

[13] Cve-2015-2716 detail. https://nvd.nist.gov/vuln/detail/CVE-2015-2716.

[14] Cve-2016-4472 detail. https://nvd.nist.gov/vuln/detail/CVE-2016-4472.

[15] Cve-2019-10098 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-10098.

[16] Cve-2020-1927 detail. https://nvd.nist.gov/vuln/detail/CVE-2020-1927.

[17] Datadog: Modern monitoring & analytics. https://www.datadoghq.com/.

[18] DataDog Release 6.6.0. https://github.com/DataDog/datadog-agent/releases/tag/6.6.0.

[19] Debian live. https://wiki.debian.org/DebianLive.

[20] Docker Swarm overview. https://docs.docker.com/swarm/overview/.

[21] Dynamic admission control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[22] ExpensiveWall: A Dangerous Packed Malware On Google Play. https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/.

[23] Fedora servers breached after external compromise. http://www.theregister.co.uk/2011/01/25/fedora_server_compromised/.

[24] Gentoo linux security announcement 200312-01. https://archives.gentoo.org/gentoo-announce/message/7b0581416ddd91522c14513cb789f17a.

[25] Git. https://git-scm.com/.

[26] Git signed push. http://thread.gmane.org/gmane.comp.version-control.git/255520.

[27] Github. https://github.com.

[28] Go in-toto verification. https://github.com/in-toto/in-toto-golang.

[29] Grafeas + in-toto. https://github.com/in-toto/totoify-grafeas.

[30] Home: Software package data exchange. https://spdx.org/.

[31] in-toto Java. https://github.com/in-toto/in-toto-java.

[32] in-toto Jenkins plugin. https://plugins.jenkins.io/in-toto.

[33] in-toto jenkins plugin. https://github.com/jenkinsci/in-toto-plugin.

[34] in-toto layout creation tool. https://in-toto.engineering.nyu.edu.

[35] in-toto-layout-tool: tooldb. https://github.com/in-toto/layout-web-tool/blob/develop/tooldb.py.

[36] in-toto Metadata Examples. https://in-toto.github.io/metadata-examples.html.

[37] in-toto openSUSE demo. https://github.com/in-toto/demo-opensuse.

[38] in-toto PyPi. https://pypi.org/project/in-toto.

[39] in-toto Specification: Version 0.9. https://github.com/in-toto/docs/blob/v0.9/in-toto-spec.md.

[40] in-toto Specifications. https://in-toto.github.io/specs.html.

[41] in-toto transport for apt. https://github.com/in-toto/apt-transport-in-toto.

[42] in-toto-webhook. https://github.com/SantiagoTorres/in-toto-webhook.

[43] Installer/development. https://wiki.ubuntu.com/Installer/Development.

[44] Jenkins: Build great things at any scale. https://jenkins.io/.

[45] Kali — penetration testing and ethical hacking linux distribution. https://www.kali.org/.

[46] Kernel.org Linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/.

[47] Kubesec.io: Quantify risk for kubernetes resources. https://kubesec.io/.

[48] Managing releases in a github repository. https://help.github.com/en/github/administering-a-repository/managing-releases-in-a-repository.

[49] The missing package manager for macos (or linux). https://brew.sh/.

[50] Notary. https://docs.docker.com/samples/library/notary/.

[51] Npm: build amazing things. https://www.npmjs.com/.

[52] NTIA: Software Component Transparency. https://www.ntia.doc.gov/SoftwareTransparency.

[53] Ocaml package manager. https://opam.ocaml.org/.

[54] Oci image format specification. https://github.com/opencontainers/image-spec.

[55] OpenChain. https://www.openchainproject.org/.

[56] Operation Aurora. https://en.wikipedia.org/wiki/Operation_Aurora.

[57] Oss fuzz. https://google.github.io/oss-fuzz/.

[58] Pacman-5.1 – Don't Use the Force, Luke! http://allanmcrae.com/2018/05/pacman-5-1-dont-use-the-force-luke/.

[59] Pacman: A simple library based package manager. https://archlinux.org/pacman/.

[60] pacman-dev: check for invalid tags. https://lists.archlinux.org/pipermail/pacman-dev/2017-September/022123.html.

[61] Plot to steal cryptocurrency foiled by the npm security team. https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm.

[62] Production-Grade Container Orchestration. https://kubernetes.io/.

[63] Python Wheels. https://pythonwheels.com/.

[64] Redhat: we make open source technologies for the enterprise. https://www.redhat.com/en.

[65] Reproducible builds. https://reproducible-builds.org/.

[66] Reproducible builds: Who is involved? https://reproducible-builds.org/who/.

[67] Security/binary transparency. https://wiki.mozilla.org/Security/Binary_Transparency.

[68] Some Debian Project machines compromised. https://www.debian.org/News/2003/20031121.

[69] Sparts. https://github.com/hyperledger-labs/SParts.

[70] The YubiKey. https://www.yubico.com/products/yubikey-hardware/.

[71] Tool-to-Tool Software Bill of Materials Exchange. https://www.it-cisq.org/software-bill-of-materials/.

[72] toto-pip. https://github.com/in-toto/toto-pip.

[73] Twistlock: Cloud Native Security for Docker, Kubernetes and Beyond. https://www.twistlock.com/.

[74] The update framework (tuf). https://theupdateframework.github.io/.

[75] Upstream (software development).

[76] Webmin 1.890 exploit - what happened? http://www.webmin.com/exploit.html.

[77] Wikipedia: Live cd. https://en.wikipedia.org/wiki/Live_CD.

[78] Yast. https://yast.opensuse.org/.

[79] Yum: Yellowdog's updated modified. http://yum.baseurl.org/.

[80] Grafeas. https://grafeas.io/, 2017.

[81] Forbes cloud 100: #19 datadog, 2018. https://www.forbes.com/companies/datadog/?list=cloud100#3cad45279e03.

[82] in-toto at the reproducible builds summit-paris 2018, 2019. https://ssl.engineering.nyu.edu/blog/2019-01-18-in-toto-paris.

[83] *PHP PEAR Software Supply Chain Attack*, 2019. https://blog.dcso.de/php-pear-software-supply-chain-attack/.

[84] Reproducible builds: Weekly report #196, 2019. https://reproducible-builds.org/blog/posts/196/.

[85] 365 Git. Adding a GPG key to a repository. http://365git.tumblr.com/post/2813251228/adding-a-gpg-public-key-to-a-repository.

[86] A. Cherepanov. *Analysis of TeleBots' cunning backdoor*. https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor.

[87] Al Cutter Adam Eidjenberg, Ben Laurie. Verifiable data structures. https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf.

[88] Paolo Andreetto, Sergio Andreozzi, Giuseppe Avellino, Stefano Beco, Andrea Cavallini, Marco Cecchi, Vincenzo Ciaschini, Alvise Dorise, Francesco Giacomini, Alessio Gianelle, et al. The glite workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062007. IOP Publishing, 2008.

[89] Andy Greenberg. *MacOS Update Accidentally Undoes Apple's "Root" Bug Patch*. https://www.wired.com/story/macos-update-undoes-apple-root-bug-patch/.

[90] Apache. Apache Subversion. https://subversion.apache.org/.

[91] Apache Infrastructure Team. apache.org incident report for 8/28/2009. https://blogs.apache.org/infra/entry/apache_org_downtime_report, 2009.

[92] Apache Infrastructure Team. apache.org incident report for 04/09/2010. https://blogs.apache.org/infra/entry/apache_org_04_09_2010, 2010.

[93] Apple. App store. find the apps you love. and the ones you're about to. https://www.apple.com/ios/app-store/.

[94] Apple Computers. iOS Security Guide, 2016. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[95] Brad Arkin. Adobe to Revoke Code Signing Certificate. https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html, 2012.

[96] Ars Technica. “flame malware was signed by rogue ca certificate”. http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/.

[97] Ars Technica. Lenovo pcs ship with man-in-the-middle adware that breaks https connections. http://arstechnica.com/security/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/.

[98] arsTechnica. *Meet “Great Cannon”, the man-in-the-middle weapon China used on GitHub*. https://arstechnica.com/security/2015/04/meet-great-cannon-the-man-in-the-middle-weapon-china-used-on-github/.

[99] Ruediger Bachmann and Achim D Brucker. Developing secure software. *Datenschutz und Datensicherheit*, 38(4):257–261, 2014.

[100] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 631–636. IEEE Computer Society, 2010.

[101] Barb Darrow. *Adobe source code breach; it’s bad, real bad*. https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad.

[102] Marc-Elian Bégin, Guillermo Diez-Andino Sancho, Alberto Di Meglio, Enrico Ferro, Elisabetta Ronchieri, Matteo Selmi, and Marian Żurek. Build, configuration, integration and testing tools for large software projects: Etics. In *Rapid Integration of Software Engineering Techniques*, pages 81–97. Springer, 2006.

[103] Beta News. Has SSL become pointless? Researchers suspect state-sponsored CA forgery. http://betanews.com/2010/03/25/has-ssl-become-pointless-researchers-suspect-state-sponsored-ca-forgery/.

[104] bleepingcomputer. *Backdoored Python Library Caught Stealing SSH Credentials*, 2018. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/.

[105] Briarproject. Development Workflow. https://code.briarproject.org/akwizgran/briar/wikis/development-workflow.

[106] Bundler.io. Bundler: the best way to manage your application’s GEMS. http://bundler.io/git.html.

[107] Canonical. Bazaar. https://bazaar.canonical.com/en/.

[108] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. Package management security.

[109] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 565–574. ACM, 2008.

[110] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. P. Weinmann, E. Rescorla, and H. Shacham. A Systematic Analysis of the Juniper Dual EC Incident. In *Proc. of ACM CCS '16*, 2016.

[111] Stephen Checkoway, Shaanan Cohney, Christina Garman, Matthew Green, Nadia Heninger, Jacob Maskiewicz, Eric Rescorla, Hovav Shacham, and Ralf-Philipp Weinmann. A systematic analysis of the juniper dual ec incident. Cryptology ePrint Archive, Report 2016/376, 2016. http://eprint.iacr.org/.

[112] Richard Chirgwin. *Microsoft deletes deleterious file deletion bug from Windows 10*. https://www.theregister.co.uk/2018/10/10/microsoft_windows_deletion_bug/.

[113] Alex Chitu. The Android Bug 8219321. https://googlesystem.blogspot.com/2013/07/the-8219321-android-bug.html#gsc.tab=0, 2013.

[114] Chocolatey. Chocolatey software — the package manager for windows. https://chocolatey.org/.

[115] Christian Nutt. *Cloud source host Code Spaces hacked, developers lose code*. http://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.

[116] Catalin Cimpanu. Backdoor found in webmin, a popular web-based utility for managing unix servers. https://www.zdnet.com/article/backdoor-found-in-webmin-a-popular-web-based-utility-for-managing-unix-servers/.

[117] Catalin Cimpanu. *Microsoft Discovers Supply Chain Attack at Unnamed Maker of PDF Software*, 2018. https://www.bleepingcomputer.com/news/security/microsoft-discovers-supply-chain-attack-at-unnamed-maker-of-pdf-software/.

[118] Codeship. Continuous Delivery with Codeship: Fast, Secure, and fully customizable. https://codeship.com/.

[119] Colin Walters. Github: cgwalters/git-evtag. https://github.com/cgwalters/git-evtag.

[120] Lucian Constantin. Asus users fall victim to supply chain attack through backdoored update. https://www.csoonline.com/article/3384259/asus-users-fall-victim-to-supply-chain-attack-through-backdoored-update.html.

[121] Context Threat Intelligence. *Threat Advisory: The Monju Incident*, 2014. https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2014/The_Monju_Incident.pdf.

[122] Mark Coppock. *Windows Update not working after October 2018 patch? Here's how to fix it* . https://www.digitaltrends.com/computing/windows-update-not-working/.

[123] Jonathan Corbet. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[124] Jonathan Corbet. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[125] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, August 2014. USENIX Association.

[126] CrowdStrike. *Securing the supply chain*. https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf.

[127] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[128] Daniel. Update on the agama vulnerability. https://komodoplatform.com/update-agama-vulnerability/.

[129] David A. Wheeler. Software Configuration Management Security. http://www.dwheeler.com/essays/scm-security.html.

[130] Debian. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202, 2003.

[131] Debian. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[132] Delicious Brains. Install wordpress site with Git. https://deliciousbrains.com/install-wordpress-subdirectory-composer-git-submodule/.

[133] Dennis Fisher. *Researcher Finds Tor Exit Node Adding Malware to Binaries*. https://threatpost.com/researcher-finds-tor-exit-node-adding-malware-to-binaries/109008/.

[134] Premkumar T Devanbu, Philip WL Fong, and Stuart G Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 126–135. IEEE Computer Society, 1998.

[135] Dona Sarkar. *A note about the unintentional release of builds today*. https://blogs.windows.com/windowsexperience/2017/06/01/note-unintentional-release-builds-today/.

[136] Ruian Duan. *TOWARD SOLVING THE SECURITY RISKS OF OPEN-SOURCE SOFTWARE USE*. PhD thesis, Georgia Institute of Technology, 2019.

[137] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

[138] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[139] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[140] Edward Iskra. *Vulnerable Wallets and the Suspicious File*, 2017. https://bitcoingold.org/vulnerable-wallets/.

[141] Robert J Ellison, John B Goodenough, Charles B Weinstock, and Carol Woody. Evaluating and mitigating software supply chain security risks. 2010.

[142] ESET Research. *OSX/Keydnap spreads via signed Transmission application*. https://www.welivesecurity.com/2016/08/30/osxkeydnap-spreads-via-signed-transmission-application/.

[143] Extreme Tech. GitHub Hacked, millions of projects at risk of being modified or deleted. http://www.extremetech.com/computing/120981-github-hacked-millions-of-projects-at-risk-of-being-modified-or-deleted.

[144] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design & Implementation (OSDI '10)*, 2010.

[145] Felix Glaser . Exploring container security: Digging into Grafeas container image metadata. https://cloud.google.com/blog/products/gcp/exploring-container-security-digging-into-grafeas-container-image-metadata.

[146] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9. ACM, 2014.

[147] Archlinux Forums. Numix gnome 3.20. https://bbs.archlinux.org/viewtopic.php?id=211164.

[148] Jay Freeman. Yet Another Android Master Key Bug. http://www.saurik.com/id/19, 2014.

[149] Paul W. Frields. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[150] Geek.com. Major Open Source Code Repository Hacked for months, says FSF. http://www.geek.com/news/major-open-source-code-repository-hacked-for-months-says-fsf-551344/.

[151] Gentoo Linux. *Incident Reports/2018-06-28 Github*. https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github.

[152] Gentoo Linux. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[153] Mike Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. http://mikegerwitz.com/papers/git-horror-story.

[154] Git SCM. Signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

[155] Git SCM. Signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

[156] GitHub, Inc. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[157] gitolite. git as a deployment tool. http://gitolite.com/deploy.html.

[158] GNU Savannah. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[159] Dan Goodin. Two new supply-chain attacks come to light in less than a week. https://arstechnica.com/information-technology/2018/10/two-new-supply-chain-attacks-come-to-light-in-less-than-a-week.

[160] Dan Goodin. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/, 2013.

[161] Google. Binary Authorization. https://cloud.google.com/binary-authorization/.

[162] AMR GReAT. Operation shadowhammer. https://securelist.com/operation-shadowhammer/89992/.

[163] Volker Gruhn, Christoph Hannebauer, and Christian John. Security of public continuous integration services. In *Proceedings of the 9th International Symposium on Open Collaboration*, page 15. ACM, 2013.

[164] Mike Gunderloy. Easy Git External Dependency Management with Giternal. `http://www.rubyinside.com/giternal-easy-git-external-dependency-management-1322.html`.

[165] Hackread. *Proton malware*. `https://www.hackread.com/hackers-infect-mac-users-proton-malware-using-elmedia-player`.

[166] Toshihiro Hanawa, Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, and Mitsuhisa Sato. Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 428–433. IEEE, 2010.

[167] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, 2014.

[168] Egor Homakov. How I hacked GitHub again. `http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html`.

[169] Lin Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged ssl certificates in the wild. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 83–97, Washington, DC, USA, 2014. IEEE Computer Society.

[170] I2P. Setting trust evaluation hooks. `https://geti2p.net/en/get-involved/guides/monotone#setting-up-trust-evaluation-hooks`.

[171] Idrees Patel. *Janus Vulnerability*. `https://www.xda-developers.com/janus-vulnerability-android-apps`.

[172] Jane Silber. Notice of Ubuntu Forums breach. `https://blog.ubuntu.com/2016/07/15/notice-of-security-breach-on-ubuntu-forums`.

[173] Jeff Erickson. Inside OilRig – Tracking Iran's Busiest Hacker Crew On Its Global Rampage. `https://www.forbes.com/sites/thomasbrewster/2017/02/15/oilrig-iran-hackers-cyberespionage-us-turkey-saudi-arabia/#5415a493468a`.

[174] Jensen Beeler. Millions of Motorcyclists Hacked in VerticalScope Breach. `https://www.asphaltandrubber.com/news/verticalscope-hack/`.

[175] Jeremy Kirk. *New malware overwrites software updaters*, 2010. `https://www.itworld.com/article/2755831/security/new-malware-overwrites-software-updaters.html`.

[176] Juniper. 2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756). `https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713`, Dec. 15.

[177] Gordon Kelly. *Apple iOS 12.1.4 Fails To Fix Cellular, WiFi Problems*. `https://www.forbes.com/sites/gordonkelly/2019/02/10/apple-ios-12-1-4-problem-iphone-cellular-data-wifi-upgrade-ipad/`.

[178] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. of the 36th IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[179] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[180] Bradley M. Kuhn. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. `https://savannah.gnu.org/forum/forum.php?forum_id=2752`, 2003.

[181] Trishank Kuppusamy, Vladimir Diaz, and Justin Cappos. Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2017.

[182] Trishank Karthik Kuppusamy, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. Uptane: Securing software updates for automobiles. *14th ESCAR Europe*, 2016.

[183] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 567–581, 2016.

[184] J. Li, M. Krohn, DMazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI '04)*, 2004.

[185] Morten Linderud. Reproducible builds: Break a log, good things come in trees. Master's thesis, The University of Bergen, 2019.

[186] Steve Lipner. The trustworthy computing security development lifecycle. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 2–13. IEEE, 2004.

[187] LWN. Linux kernel backdoor attempt. `https://lwn.net/Articles/57135/`.

[188] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, 2011.

[189] Martin Brinkmann. *Attention: Some Fosshub downloads compromised.* https://www.ghacks.net/2016/08/03/attention-fosshub-downloads-compromised/.

[190] Bertus @ Medium. Cryptocurrency clipboard hijacker discovered in pypi repository. https://medium.com@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8.

[191] Mark S Merkow and Lakshmikanth Raghavan. Secure and resilient software: Requirements, test cases, and testing methods. 2011.

[192] Microsoft. Microsoft secure development lifecycle. https://www.microsoft.com/en-us/sdl/default.aspx.

[193] Microsoft. Microsoft Trusted Publishers Certificate Store. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553504(v=vs.85).aspx.

[194] Microsoft. Nuget. https://docs.microsoft.com/en-us/nuget/.

[195] Microsoft. NuGetMirror. https://www.nuget.org/packages/NuGetMirror/.

[196] Microsoft. What is Nuget. https://docs.microsoft.com/en-us/nuget/what-is-nuget.

[197] Microsoft. Windows apps – microsoft store. https://www.microsoft.com/en-us/store/apps/windows.

[198] MITRE. Mitre att&ck. https://attack.mitre.org/.

[199] Matt Mullenweg. Passwords Reset. https://wordpress.org/news/2011/06/passwords-reset/, 2011.

[200] Naked Security. Flame malware used man-in-the-middle attack against Windows Update. https://nakedsecurity.sophos.com/2012/06/04/flame-malware-used-man-in-the-middle-attack-against-windows-update/.

[201] George C Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN*, 1997.

[202] George C Necula. *Proof-carrying code. design and implementation*. Springer, 2002.

[203] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. {CHAINIAC}: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1271–1287, 2017.

[204] Inc. NPM. https://www.npmjs.com/advisories/1082.

[205] International Standards Organization. Information technology – security techniques – application security – part 1: Overview and concepts. http://www.iso.org/iso/catalogue_detail.htm?csnumber=44378.

[206] Stack Overflow. Developer survey results: 2019. https://insights.stackoverflow.com/survey/2019.

[207] Darren Pauli. icloud phishing attack hooks 39 ios apps and wechat. theregister, 2015. https://www.theregister.co.uk/2015/09/21/icloud_phishing_attack_hooks_39_ios_apps_most_popular_message_client/.

[208] PCWorld. Gauss Malware: What You Need to Know. https://www.pcworld.com/article/260735/gauss_malware_what_you_need_to_know.html.

[209] J. Pellegrini. Secrecy in concurrent version control systems. In *Presented at the Brazilian Symposium on Information and Computer Security (SBSeg 2006)*, 2006.

[210] Leonardo Pilatti, Jorge Luis, Nicolas Audy, and Rafael Prikladnicki. Software configuration management over a global software development environment: Lessons learned from a case study.

[211] The Debian Project. Debian Upstream Guide. https://wiki.debian.org/UpstreamGuide.

[212] PyPA. pip: the Python package installer. https://pip.pypa.io/en/stable/.

[213] PyPA. Python Package Index. https://pypi.org.

[214] Steve Quirolgico, Jeffrey Voas, Tom Karygianni, Christoph Michael, and Karen Scarfone. *Vetting the Security of Mobile Applications*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-163.pdf.

[215] Red Hat, Inc. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[216] Jason Robbins. Adopting open source software engineering (OSSE) practices by adopting OSSE tools. *Perspectives on free and open source software*, pages 245–264, 2005.

[217] RODRIGO ARANGUA. The security flaws at the heart of the Panama Papers. https://www.wired.co.uk/article/panama-papers-mossack-fonseca-website-security-problems.

[218] RSA Research. *Kingslayer-A Supply Chain Attack*. https://www.rsa.com/content/dam/premium/en/white-paper/kingslayer-a-supply-chain-attack.pdf.

[219] RubyGems.org. Data verification. http://blog.rubygems.org/2013/01/31/data-verification.html.

[220] RubyGems.org. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[221] B. A. Sabbagh and S. Kowalski. A socio-technical framework for threat modeling a software supply chain. *IEEE Security Privacy*, 13(4):30–39, 2015.

[222] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72. ACM, 2010.

[223] Mark Samuels. Typosquatting attack puts developers at risk from infected javascript packages. `https://securityintelligence.com/news/typosquatting-attack-puts-developers-at-risk-from-infected-javascript-packages/`.

[224] Schneier on Security. Forging SSL Certificates. `https://www.schneier.com/blog/archives/2008/12/forging_ssl_cer.html`.

[225] Sebastian Benthall, Travis Pinney, JC Herz, and Kit Plummer. An Ecological Approach to Software Supply Chain Risk Management. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 130 – 136, 2016.

[226] R.G. Shirey, K.M. Hopkinson, K.E. Stewart, D.D. Hodson, and B.J. Borghetti. Analysis of implementations to secure git for use as an encrypted distributed version control system. In *48th Hawaii International Conference on System Sciences (HICSS '15)*, pages 5310–5319, 2015.

[227] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. `https://sourceforge.net/blog/phpmyadmin-back-door/`, 2012.

[228] Jared K. Smith. Security incident on Fedora infrastructure on 23 Jan 2011. `https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html`, 2011.

[229] Steve Klabnik. *Security advisory for crates.io, 2017-09-19*. `https://users.rust-lang.org/t/security-advisory-for-crates-io-2017-09-19/12960`.

[230] Swati Khandelwal. CCleaner Attack Timeline–Here's How Hackers Infected 2.3 Million PCs. `https://thehackernews.com/2018/04/ccleaner-malware-attack.html`, 2018.

[231] Symantec. W32.Duqu: The precursor to the next Stuxnet. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf`.

[232] Symantec. W32.Stuxnet Dossier. `https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[233] Symantec Corporation. Internet threat security report, 2018. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf`.

[234] Symantec Corporation. Internet threat security report, 2019. https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf.

[235] Tails. Building a Tails Image. https://tails.boum.org/contribute/build/#index3h1.

[236] ISOIEC JTC 1SC 27 IT Security techniques. *ISOIEC 27034 2011 Information technology – Security techniques – Application security*. https://www.iso.org/standard/44378.html.

[237] The Art of Simplicity. TFS Build: Build from a tag. http://bartwullems.blogspot.com/2014/01/tfs-build-build-from-git-tag.html.

[238] The FreeBSD Project. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.

[239] The PHP Group. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.

[240] The PHP Group. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.

[241] The YAML Project. The Official YAML Web Site. https://yaml.org/, 2019.

[242] Thomas Reed. *HandBrake hacked to drop new variant of Proton malware*. https://blog.malwarebytes.com/threat-analysis/mac-threat-analysis/2017/05/handbrake-hacked-to-drop-new-variant-of-proton-malware/.

[243] Thomas Reed. XcodeGhost malware infiltrates App Store. https://blog.malwarebytes.com/cybercrime/2015/09/xcodeghost-malware-infiltrates-app-store/.

[244] Ken Thompson. Reflections on Trusting Trust. http://cm.bell-labs.com/who/ken/trust.html.

[245] ThreatPost. Certificates spoofing google, facebook, godaddy could trick mobile users. https://threatpost.com/certificates-spoofing-google-facebook-godaddy-could-trick-mobile-users/104259/.

[246] Travis CI. Travis CI – test and deploy your code with confidence. https://travis-ci.org/.

[247] Trend Micro Cyber Safety Solutions Team. *Supply Chain Attack Operation Red Signature Targets South Korean Organizations*, 2018. https://blog.trendmicro.com/trendlabs-security-intelligence/supply-chain-attack-operation-red-signature-targets-south-korean-organizations/.

[248] Trend Micro Cyber Safety Solutions Team. *New Magecart Attack Delivered Through Compromised Advertising Supply Chain*, 2019. https://blog.trendmicro.com/trendlabs-security-intelligence/new-magecart-attack-delivered-through-compromised-advertising-supply-chain/.

[249] US-CERT. "SSL 3.0 Protocol Vulnerabilty and POODLE attack". http://arstechnica.com/security/2012/06/flame-malware-was-signed-by-rogue-microsoft-certificate/.

[250] Will Verduzu. Xposed Patch for Master Key and Bug 9695860 Vulnerabilities. https://www.xda-developers.com/xposed-patch-for-master-key-and-bug-9695860-vulnerabilities/, 2013.

[251] Laurie Voss. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.

[252] Waqas. Hackers infect mac users with proton malware using elmedia player. https://hackread.com/hackers-infect-mac-users-proton-malware-using-elmedia-player.

[253] Tom Warren. *Major new iOS bug can crash iPhones*. https://www.theverge.com/2018/2/15/17015654/apple-iphone-crash-ios-11-bug-imessage.

[254] Florian Weimer. CVE-2013-6435. https://access.redhat.com/security/cve/CVE-2013-6435, 2013.

[255] Daniel A. Wheeler. "The Apple goto fail vulnerability: lessons learned". http://www.dwheeler.com/essays/apple-goto-fail.html.

[256] David A. Wheeler. Software Configuration Management (SCM) Security. http://www.dwheeler.com/essays/scm-security.html.

[257] David A Wheeler. Fully countering trusting trust through diverse double-compiling. *arXiv preprint arXiv:1004.5534*, 2010. https://arxiv.org/abs/1004.5534.

[258] Wikipedia. Wikipedia: Chicago Tylenol Murders. https://en.wikipedia.org/wiki/Chicago_Tylenol_murders.

[259] Wired. Behind iphone's critical security bug, a single bad 'goto'. http://www.wired.com/2014/02/gotofail/.

[260] Wired. *'Google' Hackers had ability to alter source code'*. https://www.wired.com/2010/03/source-code-hacks.

[261] Yan/Bcrypt. Software transparency: Part 1. https://zyan.scripts.mit.edu/blog/software-transparency/.

[262] ZDNet. Gogo in-flight wi-fi serving spoofed ssl certificates. http://www.zdnet.com/article/gogo-in-flight-wi-fi-serving-spoofed-ssl-certificates/.

[263] ZDNet. *Open-source ProFTPD hacked, backdoor planted in source code.* http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code.

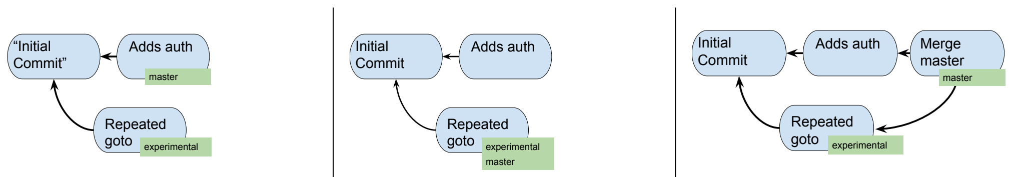
[264] ZDNet. *Red Hat's Ceph and Inktank code repositories were cracked.* http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked.

[265] ZDNet. Windows attack: Poisoned bittorrent client set off huge dofoil outbreak, says microsoft. https://zdnet.com/article/windows-attack-poisoned-bittorrent-client-set-off-huge-dofoil-utbreak-says-microsoft.

[266] ZDNet.com. Hacker explains how he put 'backdoor' in hundreds of linux mint downloads. http://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor.

Appendix A

Man In The Middle Example



(a) Original repository state (as cloned by the developer)

(b) The attacker changes the master branch pointer

(c) The developer pulled and, unknowingly, merged the experimental commit

Figure A.1: Maliciously merging vulnerable code

This appendix contains a proof of concept of a Git metadata manipulation attack against a GitHub repository with the intention of showing how an attack could be carried out in practice.

To perform an attack of this nature, an attacker controls a server, compromises a server, or acts as a man-in-the-middle between a server and a developer. Having done this, the attacker is able to provide erroneous metadata to trick a developer into committing a tampered repository state.

We simulated a repeated line scenario, in which a Git merge accidentally results a repeated line. This can be devastating as it can completely alter the flow of a program — some researchers argue that the "goto fail;" [259] vulnerability that affected Apple devices [255] might have been caused by a VCS mistakenly repeating the line while merging.

A.1 Simulating the attack

To simulate the attack, we created a repository with a minimal working sample that resembles Figure A.1c. Also, we configured two Linux machines under the same network: one functioned as the malicious server providing tampered metadata information, while the other played the role of the victim's client machine. The specific setup is described below.

Setup. To simulate the malicious server, we set up Git server on port 443 with no authentication enabled. Then, we created an SSL certificate and installed it in the victim machine. Finally, we a bare clone (using the `--bare` parameter) of the repository hosted on GitHub is created and placed on the pertinent path.

In order to redirect the user to the new branch, we modified the packed-refs file on the root of the repository so that the commit hash in the master branch matches the one for the experimental branch. Refer to Table A.1 for an example.

On the client side, a clone of the repository is created before redirecting the traffic. After cloning, the attacker's IP address is added to the victim's /etc/hosts file as "github.com" to redirect the traffic.

As such, both the server and the developer are configured to instigate the attack the next time the developer pulls.

A.2 The attack

Original file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
3a1db2295a5f842d0223088447bc7b005df86066 refs/heads/master
Tampered file
pack-refs with: peeled fully-peeled
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/experimental
00a5c1c2f52c25fe389558ea8117b7914ca2351e refs/heads/master

Table A.1: The edited packed-refs file

When the developer pulls, he or she is required to either merge or rebase the vulnerable changes into the working branch. These merged or rebased changes are not easy to identify as malicious activity, as they just resemble work performed by another developer on the same branch. Due to this, the user is likely to merge and sign the resulting merge commit.

Aftermath. Once the user successfully merges the vulnerable change, the attacker can stop re-routing the user's traffic to the malicious server. With the malicious piece of code in the local repository, the developer is now expected to pollute the legitimate server the next time he or she pushes. In this case, the attacker **was able to merge a vulnerable piece of code into production**. Even worse, there is no trace of this happening, for the target developer willingly signed the merge commit object.

Setting up an environment for this attack is straightforward; the metadata modification is easy to perform with a text editor and requires no sophistication.

Appendix B

`in-toto` artifact rule definition

The following artifact rule definition is taken from the `in-toto` specification v0.9 [39].

- `ALLOW`: indicates that artifacts matched by the pattern are allowed as materials or products of this step.
- `DISALLOW`: indicates that artifacts matched by the pattern are not allowed as materials or products of this step.
- `REQUIRE`: indicates that a pattern must appear as a material or product of this step.
- `CREATE`: indicates that products matched by the pattern must not appear as materials of this step.
- `DELETE`: indicates that materials matched by the pattern must not appear as products of this step.
- `MODIFY`: indicates that products matched by the pattern must appear as materials of this step, and their hashes must not by the same.
- `MATCH`: indicates that the artifacts filtered in using `source-path-prefix/pattern` must be matched to a ”MATERIAL” or ”PRODUCT” from a destination step with the ”`destination-path-prefix/pattern`”.

Appendix C

in-toto Integrations

- **Datadog**: in-toto has been officially integrated in the DataDog agent (version 6.6) [18].
- **Debian**: apt-transport [41] is created for in-toto, to verify Debian packages upon installation. in-toto deployment is tested through existing package manager trust infrastructure [82,84].
- **Cloud-native**: Jenkins plugin [32] and Kubernetes admission controller [21,42] have been implemented for in-toto to secure the kubesec supply chain.
- **Grafeas**: in-toto team is actively working with Grafeas to release a deployment. An ongoing implementation of this deployment is available on GitHub [29,145].
- **openSUSE**: in-toto team has released a demo to integrate in-toto and openSUSE [37].
- **Arch Linux**: From version 5.1 [58], git-tag checking (developed by in-toto team) is added to the package manager of Arch Linux, to protect against tampered git tag metadata [60].
- **CNCF**: in-toto is applying to be part of CNCF project [9].
- **Docker**: in-toto team is working on a proposal to integrate in-toto and Docker.
- **PyPi**: in-toto is available on PyPI repository [38,72].
- **Java**: A Java version of in-toto runlib is released [31].
- **Go**: A basic Go implementation of in-toto supply chain verification is available on GitHub [28].

Appendix D

Surveyed Attacks

This appendix contains a list of attacks surveyed. These attacks cover a lifespan of about 20 years and they were used to survey the results of the evaluation on Chapter 4. CR, BS and PI stand for Code Repository, Build System and Publishing Infrastructure, respectively. A ✓indicates that the attack involved a key compromise. In one attack, marked with a star (*), it was unknown if a compromised key was involved. We assumed that was the case.

Attack Name	**Key Compromise**	**Access Level**
Webmin Backdoor [76, 116]	✓	BS
purescript-npm [204]	✓	CR
electron-native-notify [61, 128]	✓	CR
ShadowHammer [120, 162]	✓	PI
PEAR breach [83]	✗	PI
DoFoil [265]	✓	PI
Operation Red [247]	✓	PI
Gentoo backdoor [151]	✗	CR
Unnamed Maker [117]	✗	PI
Colourama [159, 190]	✓	IN
HackTask [223]	✓	IN
*NotPetya [86]	✓	PI
CCleaner Attack [230]	✓	BS, PI
HandBrake breach [242]	✗	PI

Table D.2: Summary of surveyed supply chain attacks.

Attack Name	Key Compromise	Access Level
backdoored-pypi [104]	✓	PI
Bitcoin Gold breach [140]	✗	PI
Expensive Wall [22]	✗	BS
Janus Vulnerability [171]	✗	PI
OSX Elmedia Player [252]	✓	PI
Rust flaw [229]	✗	PI
Proton malware [165]	✗	PI
Buggy Windows [135]	✗	PI
Buggy Mac [89]	✗	PI
Hacked Linux Mint [266]	✗	PI
keydnap [142]	✓	PI
Juniper Incident [111]	✓	CR
FOSSHub breach [189]	✗	PI
XcodeGhost [207]	✗	BS
RedHat breach [264]	✓	BS
KingSlayer [218]	✓	PI
Monju Incident [121]	✗	PI
Code Spaces breach [115]	✗	CR
BadExit Tor [133]	✗	PI
Adobe breach [101]	✗	CR
WordPress breach [199]	✗	CR
Kernel.org breach [46]	✗	CR
Fake updater [175]	✗	PI
Google Breach [260]	✗	CR
ProFTPD breach [263]	✗	CR
Gentoo rsync [24]	✓	CR

Table D.4: Summary of surveyed supply chain attacks (cont'd)