Artemis: Defanging Software Supply Chain Attacks in Multi-repository Update Systems

Marina Moore
New York University
marinamoore@nyu.edu

Trishank Kuppusamy
Datadog
trishank.kuppusamy@datadoghq.com

Justin Cappos
New York University
jcappos@nyu.edu

ACM Reference Format:
Marina Moore, Trishank Kuppusamy, and Justin Cappos. 2023. Artemis: Defanging Software Supply Chain Attacks in Multi-repository Update Systems. In *Proceedings of ACSAC '23.* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

ABSTRACT

Modern software installation tools often use packages from more than one repository, presenting a unique set of security challenges. Such a configuration increases the risk of repository compromise and introduces attacks like dependency confusion and repository fallback. In this paper, we offer the first exploration of attacks that specifically target multiple repository update systems, and propose a unique defensive strategy we call articulated trust. Articulated trust is a principle that allows *software installation tools* to specify trusted developers and repositories for each package. To implement articulated trust, we built Artemis, a framework that introduces several new security techniques, such as per-package prioritization of repositories, multi-role delegations, multiple-repository consensus, and key pinning. These techniques allow for a greater diversity of trust relationships while eliminating the security risk of single points of failure.

To evaluate Artemis, we examine attacks on software update systems from the Cloud Native Computing Foundation's Catalog of Supply Chain Compromises, and find that the most secure configuration of Artemis can prevent all of them, compared to 14-59% for the best existing system. We also cite real-world deployments of Artemis that highlight its practicality. These include the JDF/Linux Foundation Uptane Standard that secures over-the-air updates for millions of automobiles, and TUF, which is used by many companies for secure software distribution.

1 INTRODUCTION

Software supply chain attacks are on the rise [34], with attacks more than tripling in 2021 [7] to over 30 per day [93]. One key link in the software supply chain is the software repository that distributes *packages* containing software libraries or applications to users. These repositories are often vulnerable to compromises, which can leave users vulnerable to attack [3, 4, 8, 25, 28, 32, 37–39, 65, 67, 68, 74, 77, 90, 91, 97–99, 102].

Contributing to this problem is the reality that most *software installation tools* download packages from multiple repositories. The top 10 Linux distributions come with an average of 4.8 default repositories from which users can install packages [5, 6, 71, 101]. Such multiple repository configurations introduce unique security challenges. One such challenge is vulnerability to dependency confusion attacks [87] in which attackers upload malicious packages to a public repository such as npm, PyPI, or DockerHub [30, 69, 73]. A tool that downloads packages from both a private, internal repository and a public repository may install the attacker-uploaded package from the public repository if it shares the same name as one from the internal repository. This attack is possible because software installation tools lack a mechanism to specify which repositories are to be trusted for a given package. To date, dozens of companies, including Apple, Microsoft, PayPal, Netflix, and Uber have been vulnerable to dependency confusion attacks [87]. On a broader level, the use of multiple repositories means that a key compromise can make any developer a single point of failure on a much larger scale. There have been numerous cases of developer account compromise in which benign packages were replaced with malicious ones [2, 70, 80, 86, 92, 94, 95].

Recognizing the magnitude of this threat, we present the first systematic exploration into the security of multiple repository update systems that goes beyond the trivial k of n threshold signatures [35, 85, 88]. In response to what was learned, we introduce the novel concept of *articulated trust* which enables installation tools to selectively allocate trust in repositories, projects, or developers. Articulated trust moves responsibility for indicating requirements for package installation from the repository to the installation tool by giving installation tools the control to specifically indicate which repositories and developers are trusted to provide each package.

To implement articulated trust we create Artemis, a new security framework that extends the functionality of existing Role-Based Access control (RBAC) models to multi-repository systems by incorporating a new suite of tools. These tools, including per-package prioritization of repositories, multi-role delegation, multiple-repository consensus, and key pinning, add user control and permit configuration of trusted packages. Though per-package prioritization is inspired by prioritized trust delegations [55, 61, 64] that allow a role to transfer its signing authority, Artemis permits users to define prioritized and terminating trust relationships between all entities in the update process. Multi-role delegations and multiple-repository consensus expand the existing practice of threshold, or multi-signature, signing. Instead of just requiring multiple signatures on a signed object [12, 29], it adds thresholds for other stages of the verification process. Finally, key pinning gives users more

ACSAC '23, Dec 04–08, 2023, Austin, TX

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

fine-grained control by specifying which individual developers on a repository should be trusted. Collectively, Artemis allows users to define a greater diversity of trust relationships between objects and repositories, while removing single points of failure.

Artemis was created in response to, and in partnership with, industry demand, and as a result, we were able to deploy Artemis to resolve practical security challenges. In the automotive software update system Uptane, Artemis requires consensus between repositories that provide different security properties as a way to repel efforts by nation state attackers. Among other adoptions, Uptane is incorporated into Automotive Grade Linux, an open source codebase used in millions of vehicles. Furthermore, we examined the Cloud Native Computing Foundation (CNCF)'s Catalog of Supply Chain Compromises (CSCC) and found that Artemis's security properties would protect against all historical software repository attacks.

The contributions of this paper are:

(1) We conduct the first exploration that focuses specifically on the security of multiple repository update systems and identify the shortcomings of existing systems.
(2) We introduce a threat model that specifically addresses the risks of using multiple repositories, particularly when repositories allow unknown developers to upload packages.
(3) We propose a new approach for managing trust relationships called "articulated trust" that builds on Role-Based Access Control models including RBDM1 [11], and implement this approach in Artemis, a secure software update framework.
(4) We back our claims by offering real-world examples of how our framework has been deployed to solve security challenges in multiple domains. Notably, Artemis is standardized as part of Uptane, and included in the CNCF graduated project The Update Framework (TUF).
(5) We evaluate the effectiveness of Artemis using previous attacks on software update systems and find that Artemis could have prevented all of them.

2 BACKGROUND

To provide context for our work, we first describe the software update ecosystem. We then describe RBAC and The Update Framework, both of which serve as a baseline for our Artemis mechanisms.

2.1 Software repositories and package managers

A *software repository* is a server that hosts and distributes software. When a specific version of a software application or library is ready to be released, it is built into a package, and uploaded to the repository. Three groups of people may interact with the repository. *Repository administrators* manage the software and hardware, *developers* upload packages, and *software installation tools* download, validate, and install packages with a package manager. The software installation tool is configured by a *tool administrator*, who may be a user, IT department, or package manager.

Software repositories vary in terms of their purpose and how they function. These include:

- Language-specific repositories such as PyPI, RubyGems, and npm.
- Operating system repositories, including those provided by Linux distributions. For example Arch Linux has 4 official repositories: core, extra, community, and multi-lib, each of which has a different set of packages and developers.
- Curated repositories that contain packages for a particular purpose, such as for a company's internal use. They may be a subset of other repositories, include packages from multiple sources, or contain internal packages that are not public.

A package manager is responsible for solving three problems: (1) downloading packages, (2) installing packages, and (3) performing dependency resolution [16, 21]. The latter step ensures that all required packages are downloaded, and that there are no conflicts between them.

Previous work [14, 19] has shown that it is remarkably easy for attackers to tamper with files downloaded by package managers. As such, they are vulnerable to a wide variety of threats, including arbitrary software and replay attacks. An arbitrary software attack occurs when an attacker is able to convince a software installation tool to install a malicious package instead of the intended one. Similarly, a replay attack occurs when an attacker is able to convince a tool to install an outdated, potentially vulnerable, version of a package. Even when a secure transport mechanism like TLS is used, a repository or developer key compromise can allow an attacker to manipulate update contents. Thus, package managers require protections specific to this domain. Ladisa et al. [54] describe known attacks and safeguards.

2.2 Role-Based Access Control

RBAC is a technique for restricting system access to only authorized users by associating permissions with different roles [78]. Users are assigned to roles, inheriting the associated set of permissions. RBDM0 [10] (role based delegation model) extends RBAC by allowing roles to delegate their permissions to other roles for a fixed period of time. Delegations may be revoked by the original members, or may be designated as terminating, which would prevent delegated users from further delegating this role. RBDM1 [11] builds on RBDM0 by setting up a hierarchy between roles and allowing the delegating role to give a subset of permissions to each delegated role.

The following is a list of the original RBAC96 components:

- U and R and P are sets of users, roles, and permissions, respectively
- $UA \subseteq U \times R$ is a many to many user to role assignment relation
- $PA \subseteq P \times R$ is a many to many permission to role assignment relations
- Users: $R \rightarrow 2U$ is a function derived from UA mapping each role r to a set of users where $Users(r) = \{U | (U, r) \in UA\}$
- Permissions: $R \rightarrow 2P$ is a function derived from PA mapping each role to a set of permissions where Permissions $(r) = \{P | (P, r) \in PA\}$

The RBDM0 model adds the following components:

- $UAO \subseteq U \times R$ is a many to many original member to role assignment relation
- $UAD \subseteq U \times R$ is a many to many delegate member to role assignment relation

- $UA = UAO \cup UAD$
- $UAO \cap UAD = \phi$ Original members and delegate members in the same role are disjoint
- $Users\_O(r) = \{U|(U, r) \in UAO\}$
- $Users\_D(r) = \{U|(U, r) \in UAD\}$
- All members $Users\_O(r) \cup Users\_D(r)$ in a role get all the permissions assigned to that role
- Note that $Users\_O(r) \cap Users\_D(r) = \phi$ because $UAO \cap UAD = \phi$
- T is a set of durations
- Delegate roles: $UAD \rightarrow T$ is a function mapping each delegation to a single duration

The RBDM1 model adds the following elements: $RH \subseteq R \times R$ is a partially ordered role hierarchy (this can be written as $\geq$ in infix notation). Also, the less familiar symbol $\|$ is used to denote non-comparability. We write $x \| y$ if $x\neg \leq y$ and $y\neg \leq x$.

When using RBAC in software repositories, the users are the developers, and the permissions are the set of package namespaces associated with a particular role. In this way, a role is responsible for a specific set of developers and package namespaces.

However, RBAC models are built on the assumption that this permissions model will not be tampered with. Should an attacker compromise the repository and alter the metadata that describes the RBAC permissions, they would be able to bypass these restrictions. To prevent this, researchers have created compromise-resilient software update systems [50, 79].

2.3 The Update Framework (TUF)

A compromise-resilient software update system continues to provide the correct packages even when attackers control the repository and one or more signing keys [50, 79]. The need to design such update systems was first discussed in the literature by Bellissimo et al. [14] and Cappos et al. [19]. Knockel et al. [47] observed that attackers need not even compromise a repository: a simple man-in-the-middle attack on third-party software updaters is sufficient to replace packages with malware.

The Update Framework (TUF) was designed for compromise-resilience [50, 79]. It uses different *roles* [81, 82] to sign different types of metadata, with each role requiring a *threshold* of m out of n signatures for acceptance. Thus, a single key compromise does not impact the security of the whole system. However, thresholds as a security measure only work for multiple users assigned to the same role. Roles $r \in R$ are controlled by users $u \in U$, such as repository administrators, developers, or organizations that release software.

A security feature of TUF is that it requires some roles to use offline keys. In doing so, TUF ensures that an attacker with repository access cannot compromise the entire system. Offline keys are on physical devices and inaccessible from the repository, safeguarding them in the event of a repository compromise. We use a subscript to indicate the key for each role. So $A_{\boldsymbol{k}_{off}}$ denotes role A with an offline key that is not kept on, or accessible from the repository, $B_{\boldsymbol{k}_{on}}$ denotes role B with an online key accessible from the repository, and $C_{\boldsymbol{k}}$ indicates a key that may be either online or offline.

There are four top-level roles, each of which produces and signs metadata that fulfills a specific purpose. `Root` provides a root of trust that delegates to all other roles. `Targets` provides information about images, or namespaced delegations to other targets roles. Namespaced delegations give a delegatee authority for a subset of the namespace, or set of package permissions $p \in P$, for which the delegating role is responsible. Finally, `snapshot`, introduced in a TUF variant called Mercury [50], provides bandwidth efficient consistency for the whole repository, and `timestamp` provides a heartbeat to ensure that all metadata is up-to-date and not revoked.

Delegations ensure that multiple parties do not have to share keys. To create a delegation $R_{\boldsymbol{k}} \geq A_{\boldsymbol{l}}$, metadata signed by k indicates the public key l and role name of the delegatee A, as well as what portion of R's namespace $n \in P$ the delegatee is trusted to sign. l can then sign metadata for targets in n. Diplomat [53], another TUF variant that, along with Mercury, has been integrated into the TUF specification, added prioritized and terminating delegations that ensure deterministic resolution by evaluating delegations in the order they are listed.

TUF is used in production by popular repositories, such as Docker Hub [30], Datadog [49], IBM [48], and the Python Packaging Index [51]. There are also multiple independent implementations of TUF, including those developed by Amazon [15] and Google [33].

A variant of TUF, Uptane [75], was created for use in an automotive context and has been standardized as a Joint Development Foundation project of the Linux Foundation. It uses the same RBAC principles as TUF, but the mechanisms differ in practice to account for the particular requirements of the automotive industry. However, the RBAC systems are equivalent and so for the purposes of this paper, the systems can be thought of as equivalent.

TUF addresses security concerns for users downloading software from a single repository, but does not account for the additional complexity of trust relationships between repositories. In this work, we apply the principle of compromise resilience to systems that use multiple software repositories, each with multiple developers.

3 MOTIVATION

Artemis was developed as a response to real-world concerns from software repository maintainers. We use scenarios to show the limitations of using existing software installation tools to install packages from multiple software repositories. The design requirements for Artemis were tailored to these identified problems.

Two repositories disagree on a package. Researchers discovered a vulnerability to dependency confusion attacks at PayPal due to the list of dependencies in an internal PayPal package's 'package.json' file. This list included both packages on npm and internal to PayPal. Researchers uploaded packages with the same name as the internal PayPal packages to npm, and these arbitrary packages were automatically installed [87]. When a tool installs a package `foo`, which is available on both an internal repository `A` and a public repository `B`, with different contents, it needs to determine which repository to download the package from. Many existing systems, like Pacman's repository configuration and Ubuntu's Personal Package Archive (PPA) [5, 71], address this problem by prioritizing repositories. For example by first searching in `A`, then `B`. As the roles on each repository are independent, every repository has a top-level role $r \in R$, which has permission for all namespaces $p \in P$.

Yet, the tool might not always want packages from the internal repository. The company might have an unmaintained internal

copy of a package bar, so that this package is listed on both A and B. When a vulnerability is discovered in bar and patched in the upstream version on B, the version in A does not receive this patch. In such cases, the update tool needs a mechanism to articulate that foo, which is maintained by the company, should be downloaded from A, but bar should be downloaded from B. The tool needs to prioritize the repository for each package or namespace $p \in P$.

Requirement 1: Per-package prioritization of repositories. The system must allow tools to assign each namespace $n \subset P$ to a prioritized list of repositories in order to have a specific resolution for package downloads.

Installation tools only want some packages from a repository. In 2021, an attacker hijacked the developer accounts for npm packages coa and rc, leveraging these accounts to upload malicious versions of the packages. With over 20 million weekly downloads combined [2], these packages present a significant attack surface. To counter this attack, a tool must be able to use certain packages from a public repository B while avoiding exposure to packages from untrusted or compromised developers [70, 86, 94]. Instead, tools must articulate a trusted set of packages and developers from repository B, ensuring that only these authorized packages are downloaded. When defining a set of trusted packages, tool administrators must balance the need to define trusted developers with the risk of ignoring revocation information from the repository. This tradeoff is further discussed in Section 7.2.3.

Requirement 2: Defining a trusted subset. Installation tools should be able to maintain a trusted set of roles R and developers U on a repository to ensure the installation of only trusted packages from trusted developers.

Fallback problem. If repository A is unavailable due to a networking issue or a denial of service attack, even with repository priority the tool would fallback to B to download foo. Thus, the tool needs a way to prevent falling back to other repositories. In other cases, a company might maintain repository C as a backup copy of repository A. In this case, the tool should look for foo on repository A and C, but not B. This requires the tool to specify both repository priority and when a search for a package should stop.

Requirement 3: Terminate search for a package. The installation tool can define when the per-package prioritization of repositories should stop the search for a particular namespace $n \subset P$

Repository compromise. In 2018 PEAR, the PHP Extension and Application Repository, was compromised and an installation script was replaced with a malicious version, infecting all users for 6 months before the attack was discovered [72]. Such repository compromises are common, even for prominent organizations such as Microsoft, Debian, and Apache[3, 4, 27, 28, 67]. If a repository is compromised, the attacker can replace any package or delegation signed by online keys. To mitigate this a tool may want to require that all packages from repository A match those packages on repository C. If so, the tool needs a way to specify the repositories that must reach consensus before a package is installed.

Requirement 4: Mitigate repository compromise. The installation tool should be able to require a threshold of repositories to agree on the contents of a package.

Maintainer compromise. In 2022, the author of the popular npm package node-ipc intentionally published malicious versions as a form of protest [92]. In other cases, trusted maintainers have inadvertently uploaded malicious software due to a compromise of their accounts or signing keys [2, 70, 86, 94, 95]. TUF delegations allow delegators to revoke keys for these maintainers, but only after the attack is discovered. To prevent such attacks, tools need a way to not only ensure that a package is from a trusted maintainer, but also get third-party verification of the package contents from another role, such as a security scanner. This necessitates requiring a threshold of trusted roles agree on the contents of a package.

In current systems, if a role T creates delegations $T_{\boldsymbol{k_{off}}} \geq A_{\boldsymbol{l_{on}}}$ and $T_{\boldsymbol{k_{off}}} \geq B_{\boldsymbol{m_{on}}}$, and role A's online key l is compromised, an attacker can maliciously replace a package with A's assigned namespace. Since the tool trusts T to sign this package, it will check if any delegation from T contains the package. As $T \geq A$ and A is assigned the package's namespace, the tool will install it. Instead, we want the tool to only install the package if it is signed by *both* A AND B.

Requirement 5: Mitigate role compromise. The installation tool should be able to require a threshold of roles to agree on the contents of a package.

Real-world use. Since Artemis is designed for real-world use, we also added practical requirements for smooth industry adoption.

Practical requirement 1: Shareable configuration. The end user may not be a security expert, and so should not make decisions about prioritization and threshold requirements. To address this, we ensure one expert within an organization can make configurations, then securely distribute them to all installation tools.

Practical requirement 2: Preserve backwards compatibility with existing systems. We were surprised by the extent to which industry users valued backwards compatibility. Users of existing systems do not want to install new systems, but rather want to incorporate new security mechanisms into their legacy update systems. Therefore, any proposed solution must maintain backwards compatibility with existing systems designed for a single-repository setting. Otherwise, billions of downloads from these major repositories will face serious disruptions in service. [30].

Practical requirement 3: Mechanisms added must not significantly effect performance. Any new features should minimally impact performance so that tools can easily integrate new security mechanisms. Specifically, the mechanisms should not require significant storage, download size, or computation time.

4 THREAT MODEL

As security was paramount in our design, we establish a realistic threat model for software update systems in a multi-repository setting. We assume the following actors in our system:

- **Repositories** contain software packages and online keys used to sign metadata about these packages for timeliness and consistency.
- **Developers** upload software to repositories.
- **Repository Administrators** control offline repository keys and repository configuration.
- **Software installation tools** download software from repositories.
- **Tool administrators** create configurations for software installation tools.
- **Users** request packages through software installation tools.

We assume attackers can perform all of the following actions:

(1) Respond to user requests, either by acting as a man-in-the-middle on the network, or by compromising a repository.
(2) Compromise one or more keys used to sign metadata, and hence packages, for a repository. These keys may be *online*, $m_{\boldsymbol{k_{on}}}$ or *offline*, $m_{\boldsymbol{k_{off}}}$.
(3) Use a set of keys that has been compromised to perform *arbitrary software attacks* by replacing packages whose keys have been compromised with malicious versions.
(4) Upload an arbitrary package to an unused name on a public repository.

An attack will be deemed successful if it convinces the installation tool to install a less-preferred or arbitrary package. Our goal is to achieve compromise-resilience in this setting, meaning an attacker may compromise some $u \subset U$, but *not* all, repositories or signing keys. Each compromise is bounded by the amount of time required for administrators to recover and restore the repository.

Although we focus on arbitrary software attacks, our system leverages existing software update security systems that can also address other attacks that occur on a single repository [17, 19–21].

The following problems are out of the scope of this paper, but have been discussed in other work:

- Denial-of-service attacks. TUF detects, but does not prevent denial-of-service attacks [79]. Other work has focused on denial-of-service prevention [104].
- Dependency resolution, or the problem of finding a complete set of packages that can be installed together without conflicts. Many mechanisms exist to address this issue [16, 21].
- Attacks on the software update supply chain before a package is uploaded to a repository, including source code security, continuous integration and delivery, and packaging. in-toto, which has been used with this work, can provide supply chain security [43, 49, 100].
- Remote exploits, or a compromise of users' systems through a mechanism other than software updates. Such attacks could subvert software installation tools.
- Attack detection. Artemis provides mechanisms to reduce the impact of an attack, and securely recover. However attack detection is out of scope, and can be performed by existing static and dynamic analysis and monitoring tools.

5 ARTEMIS: DESIGN

In order to address the threat model and requirements from Section 3, we extend the delegation model found in RBDM1 and TUF to implement articulated trust by adding multi-role delegations and user pinning of trusted roles. We apply this model to both roles on a repository and to the relationship between multiple repositories. An overview of the design is illustrated in fig. 1.

5.1 Multi-role delegations

First, we mitigate role compromise by introducing *multi-role* delegations. Unlike RBDM1's threshold signing that requires a threshold of keys, multi-role delegations require a threshold of roles, granting a subset of their permissions to multiple roles, but only if those roles agree. Multi-role delegations extend RBDM1 as follows:

- $UAM \subseteq U \times R$ is a many to many multi-role group to role assignment relationship.
- $UAM = UAM_1 \wedge UAM_2 \wedge ... \wedge UAM_n$ for $n \geq 1$ UAM consists of one or more sub-roles that must be in agreement on any action.
- $UAM \subseteq UAD$ Multi-role delegations are part of UAD and contain the same properties described in RBDM0 and RBDM1.

In real-world applications, the RBAC model must be stored on the repository and communicated to users who perform verification. If an attacker compromises the repository or any online keys, they would be able to tamper with the RBAC model definitions. To address this, Artemis builds its role-based model on top of TUF's delegation model, which can prevent an attacker from undermining access control through the use of offline keys and revocation.

Figure 2 conceptualizes a multi-role delegation $targets_{\boldsymbol{k_{off}}} \geq Bob_{\boldsymbol{l_{on}}} \wedge testing_{\boldsymbol{m_{on}}}$. If Bob's key 1 is compromised, the user will see that the secure hash listed by `Bob` for the malicious package does not match the non-malicious hash listed in `testing` and will abort the installation. Further, as `targets` is associated with offline keys k, either delegated role may be securely revoked. In order to replace `Ubuntu` with malware, attackers would have to compromise at least 4 keys across 2 roles.

Addresses Requirement 5: Mitigate role compromise. Multi-role delegations allow the tool to require agreement between multiple roles, preventing any role from being a single point of failure.

5.2 Key pinning

Next, we introduce a mechanism through which software installation tools can articulate a trusted subset of packages on a repository. We do so by pinning trusted roles, a process that can be defined as follows for a repository with a set of roles R:

- Installation tools define $R_U \subseteq R$, the set of roles they would like to use from the repository.
- Any A such that $(B \in R_U) \geq A$ inherits membership in R_U.

Defining R_U allows tools to locally define the roles on a repository they would like to trust, which overrides any delegations listed on the repository. This protects against malicious repository maintainers, but puts the onus on the tool to keep R_U up-to-date.

R_U is defined in the *targets map file* in Artemis. This file pins the roles R_U in a local directory alongside the installation tool. The software installation tool will use its existing verification process, but will select and use only metadata from roles R_U, thus overriding roles and keys listed by the repository. Targets metadata files for roles in R_U must be present on the repository as they must be listed

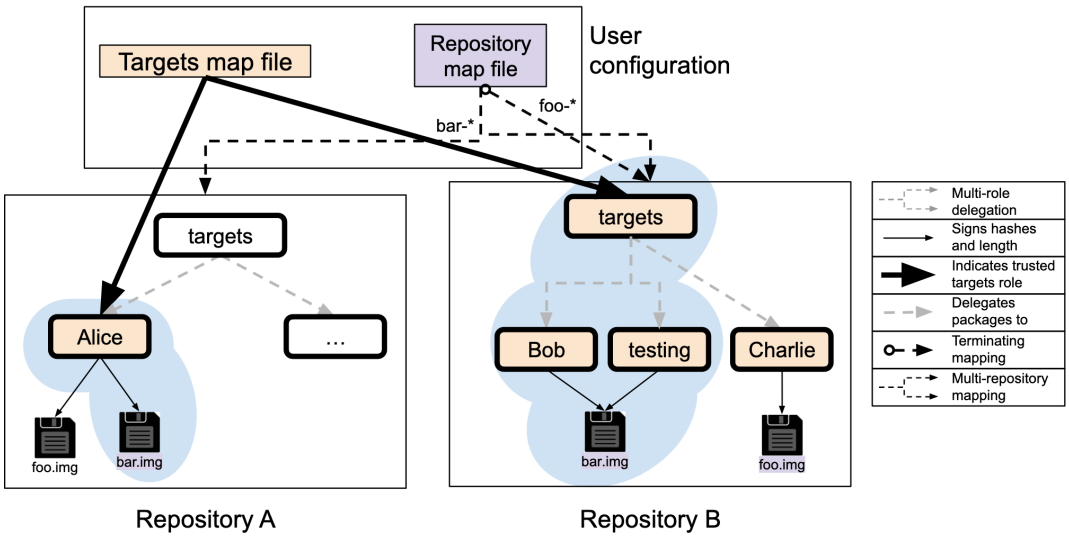

Figure 1: The overall design of Artemis. In this example, the targets map file indicates that only Alice should be trusted from repository A. The repository map file indicates that foo should be downloaded from repository B, while bar requires multiple-repository consensus from both repository A and B. On repository B, the bar image is signed by both Bob and the testing role. The roles highlighted in orange are the trusted roles indicated in the targets map file, while the images in purple are the images assigned to each repository by the repository map file. The files with a blue background represent all of the files that will be downloaded to verify bar.

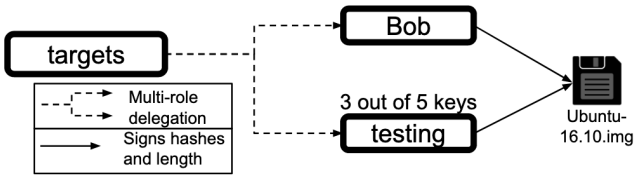

Figure 2: A multi-role delegation requires that keys and/or thresholds of keys must agree on package contents. In this example, a multi-role delegation means that `Bob` and at least 3 keys of the `testing` role must agree on the hashes and length of `Ubuntu` packages.

in its snapshot metadata. Figure 3 presents a targets map file that pins the delegated Django targets role.

One downside to pinning keys is that the tool cannot take advantage of automated key revocation from the repository. The tool defines R_U, and as a consequence these roles cannot be revoked by delegating roles on the repository. For this reason, targets map file users should ensure that they have up-to-date information about their pinned keys from tool administrators so compromised keys can be removed quickly. Tool administrators should ensure prompt updates of targets map file configurations, such as by using TUF to distribute the configuration. Tool administrators may also take advantage of delegations to update pinned keys by pinning a trusted role $I_{k_{off}}$ that can further delegate to the roles used to sign packages. $I_{k_{off}}$ is signed with offline keys, and so is more protected from a key compromise. Thus, $I_{k_{off}}$ can later be used to revoke or replace the online package signing keys.

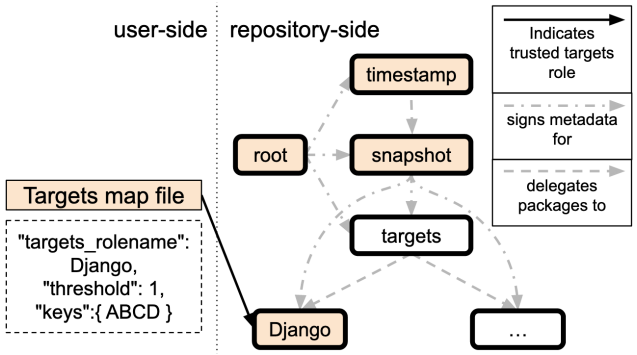

Figure 3: The targets map file pins keys for specific delegated targets roles, using the pinned roles as the top-level targets and preventing tampering by a malicious repository. In this example, a targets map file pins keys for the `Django` targets role, so that the top-level targets and root on the repository are not used to determine keys used to verify this role. During software installation, only the highlighted roles will be used, limiting the repository to packages signed by the `Django` role. The repository's root is only used to determine keys for the snapshot and timestamp roles.

For example when a user knows the keys associated with a role g_{ABC}, they may wish to reject any package that is signed with a different key EF1, even if the repository re-defines this role as g_{EF1}. Using Artemis, the user may create R_U to pin $g_{ABC} \in R_U$ so that a malicious repository cannot replace its key.

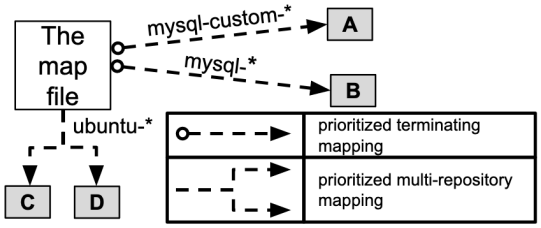

Figure 4: A repository map file provides a ordered, prioritized mapping of repositories that restricts each repository to a namespace. Mappings are prioritized in order of appearance from top to bottom. In this example, all packages starting with `mysql-custom` package are downloaded from `A` and all other `MySQL` packages are downloaded from `B`. This is combined with a multi-role delegation to repositories `C` and `D` for `ubuntu` packages. A compromise of a repository will be limited to the namespace assigned to that repository.

Addresses requirement 2: Defining a trusted subset. By allowing end users to define R_U, key pinning gives the installation tool control over the trusted roles and developers. This ensures that the repository cannot alter delegations and gives the tool granular control over key revocation. In addition, key pinning prevents a tool from automatically trusting a new, malicious developer.

5.3 Repository RBAC

Finally, articulated trust addresses per-package prioritization and mitigates the impact of repository compromise through a secure configuration of multiple repositories. We extend the application of the multi-role delegation model to apply not only to roles within a repository, but also to the relationship between repositories. We define *Repository RBAC* as an RBAC system in which the repositories are the users U_R, the namespaces are the permissions P_R and the roles R_R are assigned a set of permissions and users. We apply the multi-role delegation model to Repository RBAC to allow for the same agreement between repositories as is achieved between roles.

The *repository map file* configures Repository RBAC in Artemis. It allows tool administrators to unambiguously allocate packages to repositories, preventing dependency confusion attacks and reducing the impact of a repository compromise. Each repository map file specifies U_R, P_R, and R_R.

First, the map file contains a list of the available repositories U_R. Each repository is associated with: (1) a unique directory name where its metadata files are cached, and (2) a list of one or more URLs, each of which points to a root directory where metadata and packages are available.

Next, the map file specifies a list of *repository mappings* that define the roles R_R by associating namespaces P_R with each $u \in U_R$. Similar to prioritized delegations [53], all repository mappings are listed in a descending order of priority. Each mapping specifies: (1) a list of one or more filename patterns, (2) a list of *one or more* repositories, and (3) a flag indicating whether or not the mapping is terminating. Akin to terminating delegations [53], a terminating mapping signals to a software update security system that it should halt its backtracking search for a package (described in Section 6.1), so the user should ignore any delegation $A \geq B$ from a terminating role A. Since repository mappings are strictly ordered, there will always be one trusted conclusion for a package's metadata, or none at all if no set of trusted repositories has signed the package.

Finally, the repository map file contains a multi-role delegation threshold that specifies the number of mappings that must agree on the contents of a package. The tool will search the prioritized repository mappings until the designated threshold of mappings agrees on the package contents. This threshold gives the user protection from malicious maintainers or a repository compromise.

For repositories $A \in U_R$ and $B \in U_R$ assigned to roles R_A and R_B, respectively, Artemis can specify that namespaces $a \in P_R$ should be assigned to R_A, $b \in P_R$ should be assigned to R_B, and $c \in P_R$ should be assigned to R_A, then R_B. Thus, a package in namespace a will only be downloaded from repository A, while a package in namespace c will be looked for first in A, then in B. Figure 4 provides an example of a repository map file.

In this way the tool is able to provide per-package prioritization of multiple repositories, preventing dependency confusion attacks. Packages on trusted internal repositories cannot be replaced with arbitrary packages from a public repository, as these repositories are assigned different permissions. A tool is able to maintain a collection of verified or proprietary packages, while safely taking advantage of existing public repositories for other packages.

Addresses requirement 1: Per-package prioritization of repositories. Artemis explicitly allocates trust for each package by using prioritized repository mappings that specify which repository should be used for each namespace $p \in P_R$.

Addresses requirement 3: Terminate search for a package. The repository map file allows any assignment to a repository to be terminating, so that the tool will stop the search for a namespace n.

Further, the repository map file can create multiple repository consensus. A tool may use a multi-role delegation in Repository RBAC from the root role $r \in R_R$ for repositories $A \in U_R$ and $B \in U_R$. So $r \geq R_A \wedge R_B$. A tool will automatically reject a package if the hash of the package downloaded from repository `A` does not match that of the package downloaded from repository `B`.

Addresses requirement 4: Mitigate repository compromise. The thresholds in the repository map file allow the tool to ensure that multiple repositories agree on package contents, and thus prevent any repository from being a single point of failure.

6 IMPLEMENTATION

We integrated Artemis into TUF and Uptane, and our implementation has been upstreamed into production use. There are different implementations for each integration, using some or all of Artemis's features. These implementations include an integration in Scala for Automotive Grade Linux, as well as libraries in go and Python.

In the `python-tuf` integration, adding all features of Artemis only adds about 150 lines of code to the existing 3500 line codebase. Processing the repository map file added about 100 additional lines and the targets map file about 50 lines. This does not include additions due to unit tests, integration tests, and documentation.

Our implementation adds the repository and targets map files, which require no changes to the repository, and thus preserve backwards compatibility as specified in Section 4. As these new file types are not signed, they can be distributed by a tool administrator to a user using TUF or another secure distribution mechanism. Map file examples are shown in Appendix A.

Multi-role delegations are implemented on the repository by adding a *min_roles_in_agreement* field to delegations and allowing each delegation to list multiple roles. The user must ensure that at least *min_roles_in_agreement* roles have signed the same file hashes for a package.

6.1 Software update workflow

Artemis requires the following workflow for downloading and verifying software. First, the software update security system loads the repository map file and iterates over the list of mappings. Then each is processed using the following steps:

(1) If the filename of the desired package matches one of the paths in the list of filename patterns, then go to step 2. Otherwise, go to step 5.
(2) Download and verify the metadata for the package. The metadata will contain the hashes and length of the package.
(3) If the package hashes and length match across a threshold of repositories, then download the package, verify that it matches this metadata, stop the search, and return it.
(4) If there is no metadata about the package from any of these repositories, or if this is a terminating mapping and the package hashes and length do not match across these repositories then stop the search, and report that the package is missing.
(5) Continue to the next mapping.

In step 2, the software update security system will use its established workflow to download and verify metadata from each repository, unless the targets map file is used. In the latter case, the system will only use R_U and therefore the keys assigned to R_U. These roles may then delegate to other roles on the repository. The system will verify only packages signed by roles in R_U, or a delegatee of R_U. All other security checks are done using metadata provided by the repository.

Once it has established which repositories and keys should be used, Artemis uses a pre-order, depth-first search [53] to resolve prioritized and / or terminating delegations, and find metadata for the package. When multi-role delegations are used, Artemis modifies this search only by ensuring that the package hashes and length match across multiple, prioritized roles.

7 EVALUATION

We tested Artemis in two ways. First, we analyzed past attacks to see how Artemis can prevent or minimize assaults on software update systems. We also evaluated how the mechanisms of articulated trust contribute to achieving our security and usability goals.

7.1 Analysis of past attacks

To evaluate Artemis, we examined attacks from the CNCF's CSCC [84]. This catalog offers a cross-section of the many types of attacks on supply chains, with a particular focus on cloud applications. Of the 59 attacks cataloged before October 2022, we analyzed only those that targeted software update or distribution systems and sorted them by type of compromise. These attack types are:

- **Repository compromise**: The attacker gained control of the software repository.
- **Compromised developer key**: The attacker compromised a developer's signing key or account.
- **Compromised key and repository**: The attacker compromised both a signing key and the repository.
- **Compromised key of another trusted developer**: The attacker gained control of a trusted key for a developer other than the one who usually signs the affected package.
- **Redirect to attacker repository**: The attacker convinced users to download updates from a malicious repository
- **Malicious new developer**: A new developer joined the team or took over the project, then performed the attack.
- **Malicious existing developer**: An existing developer performed the attack.

Table 1 shows the relative effectiveness of TLS/GPG, Sigstore [89], TUF, and Artemis in preventing these attacks. Even with the usability gained from online keys, Artemis with a threshold of at least 2 roles and repositories would have prevented all 29 attacks. A configuration of Artemis without all features would still allow users to recover from all classes of attacks analyzed by revoking compromised keys.

Conversely, software update systems that rely solely on online TLS/GPG signatures would not have prevented the analyzed attacks. While TLS/GPG do provide some protection, they are so common in real-world applications [44] that recent attacks have bypassed this protection. The Sigstore project, which stores signatures on a transparent log, provides recovery from 22 analyzed attacks, but only prevents 2. Systems that use TUF with an online targets role are able to prevent 4 attacks, with high thresholds preventing a further 9. TUF is able to recover after any of the remaining 16 attacks, but could not avoid being compromised. Offline keys in TUF prevent an additional 13 attacks.

Since Artemis is designed for modular implementation, we breakdown which security properties of Artemis defend against each attack type. Key pinning, which prevented 18, allows users, rather than repositories, to specify trusted keys. Thus, key pinning prevents attacks that rely on changing trusted keys. In these attacks, an attacker manipulates the trusted keys for a package by changing a delegation. With key pinning, the user will ignore delegation changes that come from the repository or untrusted keys. This includes attacks performed by malicious new developers as key pinning allows the tool administrator to vet new developers. Multi-role delegations, which prevented 13 with online keys and 29 with offline keys, requires a threshold of at least two roles or repositories to agree on package contents, preventing a single role or repository compromise from leading to a successful attack. If a single role is compromised and provides metadata for a malicious package, the uncompromised role will not agree with anthey malicious package contents, and so the user will not install it. Finally, repository RBAC, which prevented 20, allows the user to specify which package should come from each trusted repository and developer, thus limiting the impact of a compromised developer or repository to the packages they control.

Attack Type	Count	GPG/ TLS	Sigstore	TUF		Artemis w/online targets			Artemis w/offline targets		
				Online targets	Offline targets	Key pinning	Multi-role delegations	Repository RBAC	Key pinning	Multi-role delegations	Repository RBAC
Repository compromise	13	✗	○	○	●	●	○	●	●	●	●
Compromised key and repository	3	✗	○	○	◐	◐	○	●	◐	●	●
Compromised key	6	✗	○	◐	◐	◐	●	◐	◐	●	◐
Compromised key for other trusted developer	2	✗	✗	●	●	●	●	●	●	●	●
Redirect to attacker repository	2	✗	●	●	●	●	●	●	●	●	●
Malicious new developer	1	✗	✗	◐	◐	●	●	◐	●	●	◐
Malicious existing developer	2	✗	✗	◐	◐	✗	●	✗	✗	●	✗

Table 1: Evaluating attack protection of different software update security systems, including a breakdown of protections provided by each of Artemis's features. ✗ means that the attack is not prevented and there is no way to securely recover after an attack. ○ means that the system can recover from, but not prevent an attack. ◐ means that the attack will be prevented if a threshold $t > 1$ is configured. ● means that the attack is prevented.

Not all systems that use Artemis choose the most restrictive configuration, so in practice its level of protection will depend on the chosen security properties and thresholds. For example, thresholds will only provide security gains if they are larger than one. Based on Table 1, a combination of multi-role delegations and repository RBAC could have prevented all the considered attack categories. However, this assumes that all roles and repositories have thresholds greater than one, which is not practical in all deployments. It should be noted that the property of per-package prioritization provides protection against attacks that specifically target multiple repository systems, including dependency confusion attacks, which are categorized here under redirect to attacker repository.

Further, some of the properties in Artemis are designed to work in combination. For example, multi-role delegations work best when attackers cannot replace the delegating role A. To mitigate this, a user may either pin developer keys by specifying $R_U \subseteq R$ such that $A \in R_U$, or utilize offline keys for A for additional assurance.

Adoption requirement	Deployment	Artemis features	Configured by
Define updates for each vehicle	Automotive		OEM
Protection from repository compromise	Automotive		OEM
Gather updates from multiple suppliers	Automotive		OEM
Using a third party container registry	Cloud		Package manager
Store sensitive data on a private repository	Cloud		Company
Use software from a public repository	Cloud		Package manager
Ensure updates are tested	Cloud		Package manager

Repository thresholds · Per-package prioritization · Define a trusted subset · Role thresholds · Terminate search for a package

Table 2: Different Artemis deployments use different features, often in combination to achieve specific requirements.

7.2 Real-world deployment

Artemis has been adopted in the Uptane Standard [75], which is used by automotive companies in Europe, the United States, and Japan. There are also public deployments by HERE Technologies, Airbiquity, Foundries.io, and Automotive Grade Linux [46]. Further, TUF uses Artemis in container registries, such as those used in production by DataDog, AWS, Google, Docker, and others [18]. The properties archived in these deployments are summarized in Table 2.

7.2.1 Automobiles. Artemis's multiple-repository consensus resolved Uptane's requirement to protect against nation state attackers while preserving customizability. With online keys, repositories can instantly sign different updates for different vehicles. But, online keys are much more vulnerable to compromise. Alternatively, signing software updates using offline keys provides better protection in the event of a repository compromise, but makes it harder to implement customized updates. Thus, use of offline keys can interfere with effective and time sensitive updates.

To solve this issue, Uptane uses multi-repository consensus to separate responsibility. The `Image` repository uses offline keys to sign metadata about all images. The `Director` repository uses online keys to provide instructions about which images should be installed on each vehicle. Putting these together, Artemis delegates from the root role $r \geq Director_{\boldsymbol{d_{on}}} \wedge Image_{\boldsymbol{i_{off}}}$. As the two repositories are independent of each other, attackers are unable to install malicious images unless they compromise both. This ensures that attacks on online keys are insufficient to install arbitrary packages, while allowing for on-demand customization.

Additionally, multi-role delegations allow automakers to collect updates from a number of suppliers without sharing private keys.

Key Takeaway: By using both multi-repository delegations and multi-role delegations it is now possible to ensure a software update system can remain resilient even against an actor, like a nation-state, that can compromise all repositories and online keys.

7.2.2 TUF. Artemis allows TUF to facilitate updating images from multiple container registries. A container is a form of OS-level virtualization that provides relatively isolated environments for running software. Containers are often packaged as images, or immutable snapshots of the container filesystem, hosted on repositories known as container registries, such as Docker Hub [30] and CoreOS Quay [26]. Using Artemis, users can define prioritized and terminating mappings to each registry, and specify trusted developers and packages when using an untrusted registry.

Enterprises such as eBay use Quay to distribute images to avoid the cost of hosting and maintaining its own repository. However, eBay may trust Quay for its own images, but not those of other

developers. Also, eBay does not want Quay to be able to create delegations that could replace their trusted keys. Defining a trusted subset with Artemis addresses this concern.

Per-package prioritization supports enterprise users. Enterprises cannot risk uploading images to public repositories that may expose sensitive or proprietary information. Even if access controls are placed on the images themselves, it is likely sensitive information, such as file names and public keys, will be revealed through metadata. Artemis's repository mappings allow enterprise users to store sensitive data on a private repository while taking advantage of public repositories for less sensitive images.

Key Takeaway: Key pinning and per-package prioritization allow users to trust some, but not all, packages on a registry and download these packages without trusting registry maintainers.

7.2.3 Lessons from Deployment. The initial deployment of Artemis required an investment of time from repository administrators. Systems that already store TUF metadata alongside packages require fewer changes, and so Artemis could be deployed in these systems more easily. The modular nature of Artemis allows administrators to implement features incrementally, and decide which features would provide the most security for their implementation. Uptane implemented both multi-role delegations and repository RBAC in order to support consensus between multiple roles and repositories. Conversely, users of community package repositories can implement just repository RBAC and key pinning in order to support per-package prioritization and reduce trust in the community repository.

Once Artemis is implemented, trust needs to be established in roles and repositories. Artemis uses TUF to establish trust, with the root role distributing keys for other trusted entities. This also provides mechanisms to revoke and replace trusted parties [19, 50]. TUF can further be used to distribute map files required by Artemis. A separate TUF repository with a trusted root could distribute the map files to a software installation tool. If any trusted party is later found to be malicious, TUF provides mechanisms to revoke trust. Further, role and repository thresholds in Artemis reduce the impact of any single compromised role.

Artemis requires a trusted administrator to set up and manage the security and usability tradeoffs of thresholds and pinned targets. Once these configurations are made, they can be shared among users. In Uptane, these decisions were made primarily by automotive OEMs, who determined the threshold of repositories in the repository map file, and then pushed this configuration to vehicles. In container implementations, the configuration was done by the package manager, company, or individual user that could set thresholds or pin targets based on their particular threat model. The default configuration for a package manager is shipped along with the software.

There are tradeoffs to consider when setting thresholds to balance the increased security of having more developers or repositories agree on a package's contents, and the potential for a failed update if a threshold is not met. In the case of an insufficient threshold, the tool must decide whether it is more important to meet the threshold or to install the package. For example, a tool could choose to not install the package if a signature from the original developer or a security team is missing. However, a missing signature from the QA team may be configured to not block installation. This decision must be made by looking at the purpose of each required role and the threat model of a particular application. If a threat model contains nation-state attackers that can compromise a repository or signing key, a high threshold that includes offline keys should be used. For Uptane, a threshold of two repositories allowed automakers to balance the need for real-time configuration with the need for offline keys. For applications used for hobby projects, a threshold of one may be sufficient. Community package repositories aim to make packages easy to distribute, and so would only require a single role to sign a package, removing barriers to adoption.

7.3 Usability and performance

Artemis also achieves the practical requirements from section 3.

Shareable configuration. The map files that Artemis introduces can be easily configured by experts and distributed to end-users. Map files should be distributed with TUF or a similar mechanism to ensure that the tool's copy is current and accurate. These files are designed to remain accurate as new packages are uploaded to a repository. They only need to be updated to change the trusted set of developers or policies for a package.

Backwards compatibility. As Artemis does not change the underlying techniques of TUF, it continues to prevent known attacks on single repository update systems [50, 52, 79] including replay attacks on metadata and packages. This also ensures backwards compatibility with existing TUF users. Software installation tools may add targets and repository map files as client-side configuration with no changes to repositories. However, multi-role delegations do require a change to existing metadata formats. To ensure that software installation tools do not encounter unfamiliar metadata, this change should be supported first by software installation tools, and then by developers uploading to repositories. This may be done through API versioning on the repository.

Performance. Based on performance tests of the Python implementation, Artemis adds minimal processing time to software update systems. We compare runtime and metadata sizes with and without repository and targets map files. For this test, the repository map file requires that two repositories agree on package contents. Our results are summarized in Table 3 and show only an increase of 10 milliseconds for verification. If more repositories or roles are verified, there would be more processing, but this increase is still small for users considering that downloading and installing software packages, which are often many megabytes in size, dominates the overall processing time. For example PyPI packages, based on publicly available data, average over 3MB [103].

The Python implementation of Artemis is built on top of a TUF implementation, and so has similar performance as the size of repositories grows. An analysis of the runtime and metadata overhead of TUF was performed in Mercury [50].

Artemis can enhance security protections with a manageable change to existing systems. Our implementation added only about 4% more code to the software update system. The simplicity of the additional code required for verification makes it easy to implement.

Metric	TUF	Artemis	Package download	Artemis overhead
Processing time	200 ms	210 ms	240 ms	38%
Storage	9.511 KB	10.262 KB	3 MB	0.34%

Table 3: Artemis runtime and storage. The storage for TUF is equivalent to the bandwidth. However, the additional metadata in Artemis is configured locally and not downloaded from a repository. Package download time is based on the average PyPI package at an 100 Mbps download speed.

8 RELATED WORK

We compare Artemis to the broader field of software supply chain security, as well as other work built on in this paper.

Software supply chain security Artemis improves security of software distribution and updating, a sub-problem of software supply chain security. Other technologies in this field solve related problems. in-toto [100] is an end-to-end framework that allows users to define and validate steps in a supply chain. Software bills of materials (SBOMs) [96] provide auditable information about software dependencies including Software Identification (SWID) tagging [23], Software Package Data Exchange (SPDX) [105], and CycloneDX [31]. Coppens et. al. present binary code diversification to prevent attackers from using binary diffs to reconstruct patched attacks [24]. These technologies can work in tandem with Artemis, with Artemis securely distributing both packages and metadata, like SBOMs, from other supply chain security technologies. Han et. al. introduce Sigl, a tool for detecting malicious software installer programs [41]. Artemis focuses instead on ensuring that the installer gets the intended artifact from a collection of software repositories.

Multi-role delegations. Artemis applies ideas from previous work in logic-based distributed authorization. D2LP is an authorization language in delegation logic [40, 60, 62, 63] that extends early works on trust management and authorization in distributed systems [1, 56]. D2LP also uses both the AND and OR relations in delegations, and could express the mechanisms in Artemis. However, Artemis is the first system for software updates that uses these types of delegations.

Threshold and multi-signature signing Artemis expands on the idea of signature thresholds by adding thresholds of roles or repositories. Multi-signature signing [13, 45, 66, 76, 106] efficiently allows multiple signatures on an artifact. Threshold signatures [35, 85, 88] allow multiple parties to sign an artifact by splitting the private key among all entities. Artemis does not require that the same metadata bytes are signed, but that the hashes and length of a package match across multiple roles or repositories.

Multiple repositories. Like Artemis, other systems use the idea of multiple repositories or servers, but there are some important differences. Linux software updaters `dnf` and `apt` allow users to install packages from multiple repositories as does Revere [59], which uses a self-organizing, peer-to-peer (P2P) overlay network to deliver updates. Essentially, every Revere node acts as a mirror, and may push or pull updates to or from other nodes. However, these systems do not solve the priority or fallback problems, or allow for multiple-repository consensus.

Byzantine fault-tolerant systems (BFT) use many replicas instead of a single server to execute operations [9, 22, 58, 83]. Yet, BFT systems are aiming to solve a different problem, that of guaranteeing linearizability [42] in a system made of distributed processors. In contrast, Artemis uses multiple repositories as independent sources of information that must agree with each other.

Reducing trust in a repository. Some software update systems allow users to reduce trust in a repository. For example, `apt` and `dnf` require user permission before revoking and replacing keys used to verify metadata. However, Artemis is the first system that removes the repository from the equation. Instead, Artemis allows users to specify their own keys for packages and metadata. This technique is powerful enough that it can solve other problems, such as reducing trust in mirrors.

Secure Untrusted Data Repository (SUNDR) [57] uses a set of trusted users that honestly report histories. SUNDR uses this information to detect equivocation to protect against arbitrary software attacks and forking attacks on a single repository. Artemis does not rely on a trusted set of honest users reporting history data and also supports articulated trust in multiple repositories.

Another approach to reducing trust is the use of binary transparency through publicly auditable, immutable transparent logs. Sigstore [89] provides a set of tools for signing packages and uploading these signatures to a transparent log for auditability and protection against forking attacks. But, this approach relies on third-party auditors of the log. Artemis is compatible with transparent logs, and and there are integrations of Artemis and Sigstore[89] that provide Artemis's articulated trust and revocation mechanisms in addition to the protections offered by transparent logs.

Ghosn et. al. [36] propose reducing the impact of malicious packages by using Enclosures to restrict the resources that library dependencies can access. Unlike Artemis, Enclosures require changes to programming languages. Artemis can work with existing developer workflows, and would be compatible with Enclosures.

9 CONCLUSION

In this paper we conducted the first comprehensive examination of the security of multiple repository update systems. We introduce articulated trust to enable a software update system to use multiple repositories while limiting the trust placed in them. Artemis, our implementation of articulated trust, has been successfully deployed in several large scale production environments. Artemis uses simple yet effective mechanisms to provide secure configuration of trust in challenging environments, such as automobiles and container registries. Through collaboration with industry practitioners, we show that Artemis addresses real world problems. For example, Artemis allows companies using popular cloud container sites to prevent dependency confusion attacks. It also helps automakers enhance their resilience against nation-state actors through multiple-repository consensus.

The security properties of Artemis are very effective in preventing previous attacks on software update systems, and would prevent all attacks that we analyzed, compared with 59% for TUF and 7% for Sigstore. By employing key pinning, eliminating single points of failure, and per-package prioritization, Artemis establishes a robust defense against attacks on multiple repository update systems.

REFERENCES

[1] Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept. 1993), 706–734.

[2] Aguirre, J. Npm hijackers at it again: Popular 'coa' and 'rc' open source libraries taken over to spread malware. *sonatype blog* (2021).

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

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

[5] *add-apt-repository*, 2021.

[6] ArchWiki. Official repositories. https://wiki.archlinux.org/title/Official_repositories, 2022.

[7] Argon. 2021 Software Supply Chain Security Report. Tech. rep., Argon: An Aqua Company.

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

[9] Aublin, P.-L., Mokhtar, S. B., and Quéma, V. Rbft: Redundant byzantine fault tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems* (USA, 2013), ICDCS '13, IEEE Computer Society, p. 297–306.

[10] Barka, E., and S, R. A role-based delegation model and some extensions. *Proceedings of the 23rd National Conference on Information Systems Security* (12 2000).

[11] Barka, E., and Sandhu, R. Role-based delegation model/hierarchical roles (rbdm1). pp. 396– 404.

[12] Bellare, M., and Neven, G. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, Association for Computing Machinery, p. 390–399.

[13] Bellare, M., and Neven, G. Multi-signatures in the plain public-key model and a general forking lemma. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, Association for Computing Machinery, p. 390–399.

[14] Bellissimo, A., Burgess, J., and Fu, K. Secure software updates: disappointments and new challenges. *Proceedings of USENIX Hot Topics in Security (HotSec)* (2006).

[15] Bottlerocket update infrastructure. https://github.com/bottlerocket-os/bottlerocket/tree/develop/sources/updater, 2019.

[16] Burrows, D. Modelling and resolving software dependencies. https://people.debian.org/~dburrows/model.pdf, 2005.

[17] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. H. Stork: package management for distributed VM environments. In *The 21st Large Installation System Administration Conference, LISA'07* (2007).

[18] Cappos, J., Kuppusamy, T. K., Lock, J., Moore, M., and Pühringer, L. The update framework specification. Specification, 2022.

[19] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 565–574.

[20] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. Package management security. *University of Arizona Technical Report* (2008), 08–02.

[21] Capppos, J. *Stork: Secure Package Management for VM Environments.* Dissertation, University of Arizona, 2008.

[22] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 173–186.

[23] Center, I. T. L. C. S. R. Software Identification (SWID)Tagging. Tech. rep., National Institute of Standards and Technology, 2021.

[24] Coppens, Bart and De Sutter, Bjorn and De Bosschere, Koen. Protecting your software updates. *IEEE SECURITY PRIVACY 11*, 2 (2013), 47–54.

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

[26] CoreOS, Inc. Quay Container Registry. https://quay.io/.

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

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

[29] Desmedt, Y. Society and group oriented cryptography: A new concept. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology* (Berlin, Heidelberg, 1987), CRYPTO '87, Springer-Verlag, p. 120–127.

[30] Docker Inc. Docker Hub. https://hub.docker.com/.

[31] Foundation, O. CycloneDx. https://cyclonedx.org/, 2021.

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

[33] Fuschia. Software update system. Tech. rep., 2021.

[34] Geer, D., Tozer, B., and Meyers, J. S. For good measure: Counting broken links: A quant's view of software supply chain security. *login Usenix Mag. 45* (2020).

[35] Gennaro, R., and Goldfeder, S. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 1179–1194.

[36] Ghosn, A., Kogias, M., Payer, M., Larus, J. R., and Bugnion, E. Enclosure: Language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, p. 255–267.

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

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

[39] Goodin, D. 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.

[40] Grosof, B. N. Prioritized conflict handling for logic programs. In *ILPS* (1997), vol. 97, pp. 197–211.

[41] Han, X., Yu, X., Pasquier, T., Li, D., Rhee, J., Mickens, J., Seltzer, M., and Chen, H. SIGL: Securing software installations through deep graph learning. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 2345–2362.

[42] Herlihy, M. P., and Wing, J. M. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1987), POPL '87, ACM, pp. 13–26.

[43] in-toto - a framework to secure the integrity of software supply chains. https://in-toto.io/, 2022.

[44] (ISRG)., I. S. R. G. Let's encrypt stats. https://letsencrypt.org/stats/, 2021.

[45] ITAKURA, K; NAKAMURA, K. A public-key cryptosystem suitable for digital multisignatures. *NEC research development* (1983).

[46] Joint Development Foundation Projects, LLC, Uptane Series. Adoptions. https://uptane.github.io/adoptions.html, 2020.

[47] Knockel, J., and Crandall, J. R. Protecting Free and Open Communications on the Internet Against Man-in-the-Middle Attacks on Third-Party Software: We're FOCI'd. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet* (Berkeley, CA, 2012), USENIX.

[48] Kubernetes. Case study: Ibm building an image trust service on kubernetes with notary and tuf. https://v1-18.docs.kubernetes.io/case-studies/ibm/, 2018.

[49] Kuppusamy, T. K. Secure Publication of Datadog Agent Integrations with TUF and in-toto. https://www.datadoghq.com/blog/engineering/secure-publication-of-datadog-agent-integrations-with-tuf-and-in-toto/, 2019.

[50] Kuppusamy, T. K., Diaz, V., and Cappos, J. Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In *USENIX ATC '17* (USA, 2017), USENIX Association, p. 673–688.

[51] Kuppusamy, T. K., Diaz, V., Stufft, D., and Cappos, J. PEP 458 – Securing the Link from PyPI to the End User. https://www.python.org/dev/peps/pep-0458/, 2013.

[52] Kuppusamy, T. K., Torres-Arias, S., Diaz, V., and Cappos, J. Diplomat: Using Delegations to Protect Community Repositories. Tech. Rep. TR-CSE-2016-01, Computer Science and Engineering, Tandon School of Engineering, New York University.

[53] Kuppusamy, T. K., Torres-Arias, S., Diaz, V., and Cappos, J. Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 567–581.

[54] Ladisa, P., Plate, H., Martinez, M., and Barais, O. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2023), IEEE Computer Society, pp. 1509–1526.

[55] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 265–310.

[56] Lampson, B., Abadi, M., Burrows, M., and Wobber, E. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 265–310.

[57] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 9–9.

[58] Li, J., and Maziéres, D. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 10–10.

[59] Li, J., Reiher, P., and Popek, G. J. Resilient self-organizing overlay networks for security update delivery. *Selected Areas in Communications, IEEE Journal on 22*, 1 (2004), 189–202.

[60] Li, N. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, 2000.

[61] Li, N., Feigenbaum, J., and Grosof, B. A logic-based knowledge representation for authorization with delegation. pp. 162 – 174.

[62] Li, N., Feigenbaum, J., and Grosof, B. N. A logic-based knowledge representation for authorization with delegation. In *Computer Security Foundations Workshop, 1999. Proceedings of the 12th IEEE* (1999), IEEE, pp. 162–174.

[63] Li, N., Grosof, B. N., and Feigenbaum, J. A Nonmonotonic Delegation Logic with Prioritized Conflict Handling. https://www.cs.purdue.edu/homes/ninghui/papers/old/d2lp.pdf, 2000.

[64] Li, N., Grosof, B. N., and Feigenbaum, J. A nonmonotonic delegation logic with prioritized conflict handling. https://www.cs.purdue.edu/homes/ninghui/papers/old/d2lp.pdf, 2000.

[65] Magnusson, H. The PHP project and Code Review. http://bjori.blogspot.com/2010/12/php-project-and-code-review.html, 2010.

[66] Micali, S., Ohta, K., and Reyzin, L. Accountable-subgroup multisignatures: Extended abstract. CCS '01, Association for Computing Machinery, p. 245–254.

[67] Microsoft, Inc. Flame malware collision attack explained. http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx, 2012.

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

[69] npm, Inc. npm. https://www.npmjs.com/.

[70] Overson, J. How two malicious npm packages targeted sabotaged others. https://jsoverson.medium.com/how-two-malicious-npm-packages-targeted-sabotaged-one-other-fed7199099c8, 2019.

[71] *pacman.conf*, 2021.

[72] When php went pear shaped- the php pear compromise. https://blog.cpanel.com/when-php-went-pear-shaped-the-php-pear-compromise/, 2022.

[73] Python Software Foundation. PyPI - the Python Package Index: Python Package Index. https://pypi.python.org/pypi.

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

[75] Redacted. Redacted for anonymous submission.

[76] Ristenpart, T., and Yilek, S. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. EUROCRYPT '07, Springer-Verlag, p. 228–245.

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

[78] S, R., Sandhu, E. J., Coyne, H. L. F., and Youman, C. E. Role based access control models. In *Computer* (February 1996), p. 38–47.

[79] Samuel, J., Mathewson, N., Cappos, J., and Dingledine, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 61–72.

[80] Sanders, J. Malicious libraries in package repositories reveal a fundamental security flaw. https://www.techrepublic.com/article/malicious-libraries-in-package-repositories-reveal-a-fundamental-security-flaw/, 2019.

[81] Sandhu, R. S. Role-based access control. *Advances in computers 46* (1998), 237–286.

[82] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[83] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[84] Security, C. T. Catalog of supply chain compromises. https://github.com/cncf/tag-security/tree/main/supply-chain-security/compromises, 2021.

[85] Shamir, A. How to share a secret. *Commun. ACM 22*, 11 (nov 1979), 612–613.

[86] Sharma, A. Newly found npm malware mines cryptocurrency on windows, linux, macos devices. *sonatype blog* (2021).

[87] Sharma, A. Researcher hacks over 35 tech firms in novel supply chain attack. https://www.bleepingcomputer.com/news/security/researcher-hacks-over-35-tech-firms-in-novel-supply-chain-attack/, 2021.

[88] Shoup, V. Practical threshold signatures. In *Advances in Cryptology — EUROCRYPT 2000* (Berlin, Heidelberg, 2000), B. Preneel, Ed., Springer Berlin Heidelberg, pp. 207–220.

[89] Sigstore. A new standard for signing, verifying and protecting software. https://www.sigstore.dev/, 2021.

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

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

[92] Snyk. CVE-2022-23812. https://nvd.nist.gov/vuln/detail/CVE-2022-23812, 2022.

[93] Socket - secure your javascript supply chain. https://socket.dev/, 2022.

[94] SuperOleg39. Security issue: compromised npm packages of ua-parser-js (0.7.29, 0.8.0, 1.0.0) - questions about deprecated npm package ua-parser-js. https://github.com/faisalman/ua-parser-js/issues/536, 2021.

[95] Tal, L., and Josef, A. B. Open source maintainer pulls the plug on npm packages colors and faker, now what? *snyk blog* (2022).

[96] Telecommunications, N., and Administration, I. Software Bill of Materials. https://www.ntia.gov/SBOM, 2021.

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

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

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

[100] Torres-Arias, S., Nanize, H., Kuppusamy, T., Curtmola, R., and Cappos, J. in-toto: providing farm-to-table security properties for bits and bytes. In *28th USENIX Security Symposium* (2019), USENIX Sec'19.

[101] Ubuntu Sources List Generator, 2018. https://repogen.simplylinux.ch/index.php.

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

[103] Warehouse. Bigquery datasets. https://warehouse.pypa.io/api-reference/bigquery-datasets.html, 2022.

[104] Wood, P., Gutierrez, C., and Bagchi, S. Denial of service elusion (dose): Keeping clients connected for less. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)* (2015), pp. 94–103.

[105] Workgroup, S. The Software Package Data Exchange. Tech. rep., The Linux Foundation, 2021.

[106] Đc Phong, L., Bonnecaze, A., and Gabillon, A. Multisignatures as secure as the diffie-hellman problem in the plain public-key model. *Lecture Notes in Computer Science 5671* (08 2009), 35–51.

```
"targets_mappings": [
   {
     // This mapping applies to all packages on PyPI.
     // This field may be omitted for single repository setups.
     "repositories": ["PyPI"],
     // The package manager will treat the targets metadata for Ebay from PyPI
     // as the top-level targets, and will not install packages from any other
     // developers.
     "targets_rolename": "Ebay",
     // Only one key is required to sign the Ebay targets metadata
     "threshold": 1,
     // This dictionary lists the keys associated with the Ebay role. These keys
     // are pinned and can only be changed by updating this configuration.
     "keys":{
         0 : ABCD
     }}]
```

Figure 5: An example of a targets map file.

```
{
  // For each repository, its key name is the directory where files, are cached
  // and its value is a list of URLs where files may be downloaded.
  "repositories": {
     "Django": ["https://djangoproject.com/"],
     "PyPI":   ["https://pypi.python.org/"]
  },
  // Specify a list of repositories where each set of targets may be downloaded.
  "mapping": [
     {
       // The order of these entries indicates the priority of the delegation.
       // The entries listed first will be considered first.

       // Map the targets "/django/django-1.*.tgz" to both Django and PyPI.
       "paths":        ["/django/django-1.*.tgz"],
       "repositories": ["Django", "PyPI"],

       // At least one repository must sign for the same length and hashes
       // of the "/django/django-1.*.tgz" targets.
       "threshold": 1

       // In this case, the "terminating" attribute is set to false.
       "terminating":  false,
       // Therefore, if this mapping has not signed for "/django/django-1.*.tgz"
       // targets, the following mapping will be consulted.
     },
     {
       // Map all other targets only to PyPI.
       "paths":        ["*"],
       "repositories": ["PyPI"],
       "threshold": 1,
       "terminating": true
     }]}
```

Figure 6: An example of a repository map file.

A MAP FILE EXAMPLES

We include examples of targets and repository map files in Figure 5 and Figure 6. These files are written and updated by tool administrators to set policy that will be applied to many update cycles.