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

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, 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
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
A compromise-resilient software update system continues to provide information about images, or namespaces, 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 delegatee is responsible. Finally, snapshot, introduced in a TUF variant called Mercury [50], provides bandwidth efficient consistency for the whole repository, and timesnap 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 \( R_k \geq A_I \), metadata signed by \( k \) indicates the public key \( I \) 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. \( I \) 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.

**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-1pc 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 \( Tk_{\text{off}} \geq A_{\text{on}} \) and \( Tk_{\text{off}} \geq B_{\text{on}} \), and role A’s online key \( i \) 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 impact 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, offline, or a combination of both.
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.

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.
- \( UAM = UAM_1 \land UAM_2 \land \ldots \land UAM_n \) 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. All 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 \( (B \in R_U) \Rightarrow A \) 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 \textit{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...
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 $l_{k_{off}}$ that can further delegate to the roles used to sign packages. $l_{k_{off}}$ is signed with offline keys, and so is more protected from a key compromise. Thus, $l_{k_{off}}$ can later be used to revoke or replace the online package signing keys.

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.

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.
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.

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 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 of code to the existing 3500 line codebase. 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 $r \geq R_A \land R_B$. 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 $B$.

Further, the repository map file allows any assignment to a repository to be terminating, so that the tool will stop the search for a namespace $n$.

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),...
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 go to step 7.
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 15 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 an they 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.
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.

### 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 $r \geq \text{Director}_\text{delegated} \land \text{Image}_\text{offline}$ 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

<table>
<thead>
<tr>
<th>Attack Type</th>
<th>Count</th>
<th>GPG</th>
<th>Sigstore</th>
<th>TLS</th>
<th>Online targets</th>
<th>Offline targets</th>
<th>TUF</th>
<th>Key pinning</th>
<th>Multi-role delegations</th>
<th>Repository RBAC</th>
<th>Artemis w/online targets</th>
<th>Artemis w/offline targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository compromise</td>
<td>13</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Compromised key and repository</td>
<td>3</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Compromised key</td>
<td>6</td>
<td>×</td>
<td>○</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Compromised key for other trusted developer</td>
<td>2</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Redirect to attacker repository</td>
<td>2</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Malicious new developer</td>
<td>1</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Malicious existing developer</td>
<td>2</td>
<td>×</td>
<td>×</td>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

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.

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

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