Diplomat: Using Delegations to Protect Community Repositories*

Trishank Karthik Kuppusamy Santiago Torres-Arias Vladimir Diaz Justin Cappos
Tandon School of Engineering, New York University

Abstract

Community repositories, such as Docker Hub, PyPI, and RubyGems, are bustling marketplaces that distribute software. Even though these repositories use common software signing techniques (e.g., GPG and TLS), attackers can still publish malicious packages after a server compromise. This is mainly because a community repository must have immediate access to signing keys in order to certify the large number of new projects that are registered each day.

This work demonstrates that community repositories can offer compromise-resilience and real-time project registration by employing mechanisms that disambiguate trust delegations. This is done through two delegation mechanisms that provide flexibility in the amount of trust assigned to different keys. Using this idea we implement Diplomat, a software update framework that supports security models with different security / usability tradeoffs. By leveraging Diplomat, a community repository can achieve near-perfect compromise-resilience while allowing real-time project registration. For example, when Diplomat is deployed and configured to maximize security on Python's community repository, less than 1% of users will be at risk even if an attacker controls the repository and is undetected for a month. Diplomat is being integrated by Ruby, CoreOS, Haskell, OCaml, and Python, and has already been deployed by Flynn, LEAP, and Docker.

1 Introduction

Community repositories, such as Docker Hub [32], Python Package Index (PyPI) [66], RubyGems [68], and SourceForge [78] provide an easy way for a developer to disseminate software. These repositories are run by a central group of administrators and distribute third-party software for hundreds of thousands of projects. Unlike traditional repositories, the administrators of community repositories do not dictate which projects can or cannot be hosted; instead, developers are free to curate their own projects. Community repositories are immensely popular and collectively serve more than a billion packages per year. Unfortunately, the popularity of these repositories also makes them an attractive target to attackers.

Attacks on community repositories are unfortunately a common occurrence that threaten users who rely on their software. Major repositories run by Adobe, Apache, Debian, Fedora, FreeBSD, Gentoo, GitHub, GNU Savannah, Linux, Microsoft, npm, Opera, PHP, RedHat, RubyGems, SourceForge, and WordPress repositories have all been compromised at least once [4,5,7,27,28,30, 31,35,36,39–41,48,59,61,62,67,70,79,80,82,86,87,90]. For example, a compromised SourceForge repository mirror located in Korea distributed a malicious version of phpMyAdmin, a popular database administration tool [79]. The modified version allowed attackers to gain system access and remotely execute PHP code on servers that installed the software. This is despite the use of off-the-shelf solutions like TLS and GPG, which (for reasons described in Section 4) are known to be ineffective against practical threats in this domain. For example, we found that, on PyPI, so few developers sign packages and so few users download signatures that within a one month period there was not a single user who downloaded only GPG-signed packages and their signatures.

Prior work has shown that *delegations* [1, 52, 92] help the users of a repository remain secure even if it is compromised [71]. Delegations add security to repositories when the root of trust is an *offline key*, such as a key stored on a disconnected server that must be manually used. Although using offline keys works for software repositories that have infrequent release cycles, community repositories commonly register dozens of new projects daily, with new packages uploaded every few minutes. As such, it is not practical to require manual operations for project registration.

This paper presents Diplomat, a practical security system that provides a community repository with immediate project registration and compromise-resilience. Our key insights come from delegation techniques that utilize multiple online and offline keys to take advantage of the best properties of both. Central to this strategy is the use of a *prioritized delegations* [44, 56–58] mechanism for disambiguating trust statements. Prioritized delegations enforce an order among parties who would otherwise be equally trusted. In addition, our work uses *terminating delegations*, which prevent statements by less trusted

*This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

parties from being trusted for a package. The combination of prioritized and terminating delegations allows an offline key's attestation about a project to be trusted over information provided by an online key. Placing greater trust in the offline key provides compromise-resilience because an attacker who compromises the repository cannot modify a package without being detected. However, the online key may still be used (and trusted) to create new projects.

We feel one of the main contributions of this work is how it balances security and usability to solve a practical, widespread problem. The security models and experiences we describe in this work are based upon practical lessons learned from ongoing integrations with RubyGems [75–77], Haskell [91], CoreOS [64], and OCaml [38] and production use in Flynn [65], LEAP (Bitmask) [53], and Docker [63].

Contributions.

- We examine threats to community repositories and find that current security approaches inadequately address these threats. In particular, these techniques are unable to accommodate both compromise-resilience and instant registration of new projects.
- We use two types of delegations — prioritized and terminating delegations — to design and implement Diplomat, the first security system that achieves both compromise-resilience and instant registration of new projects.
- We discuss two different security models — legacy and maximum — that provide slightly different usability / security trade-offs. Drawing on practical experience, we discuss procedures for managing offline key storage, usability for users and developers, recovering from compromises, and procedures to minimize the effort required of repository administrators.
- We evaluate the effectiveness of Diplomat using requests to PyPI, the main Python community repository. Our findings demonstrate that Diplomat will protect over 99% of PyPI's users, even if an attacker controls PyPI and is undetected for a month.

2 Background

We first discuss and define community repositories, paying attention to how they differ from traditional software repositories. We then provide some background on roles and delegations, two techniques used in compromise-resilient repositories [20–23, 71] that we will leverage to build Diplomat.

2.1 Community Repositories

A *community repository* hosts and distributes third-party software. Three groups of people, administrators, developers and users, interact with a community repository. The administrators, who are usually volunteers, manage the community repository software and hardware. Developers upload software to the repository, which is requested by users. Users install and validate software using a package manager, which downloads software through middlemen, such as content delivery networks and / or mirrors to reduce bandwidth costs.

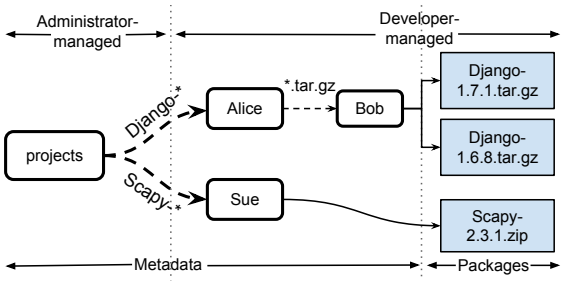

Figure 1: An example of delegation of trust in a software repository. The top-level `projects` role delegates to Sue for the Scapy project and Alice for Django. Alice further delegates to Bob the ability to create tar.gz packages for Django.

The software that is uploaded by developers is organized as follows. A developer registers a *project* with a unique name and adds access to other developers that work on the project. When a specific version of the software for that project is ready to be released, the software is built into a *package* (e.g., `Django-1.7.tar.gz`) and one of the developers uploads that package to the community repository. The community repository also distributes metadata about projects and packages (such as a list of package names) and includes metadata created by developers (such as a signature for a package).

2.2 Roles

One of the key security concepts used in compromise-resilient software repositories is that of a *role* [71]. A role defines the set of actions that a party is allowed to perform. For example, the `projects` role is trusted to sign metadata that indicates which developer keys belong to a project. Similarly, the `release` role is trusted to sign metadata that indicates which versions of each package and metadata are in the latest release. However, if the `release` role's key is used to sign the metadata that indicates which developer key belongs to a project, that signature will not be trusted because the key is not trusted for that role. This paper describes techniques that apply to the `projects` role's use of delegations, so the paper will focus primarily on this role.

2.3 Delegations

The use of *delegations* is a powerful strategy that has successfully been used in a variety of contexts, including distributed systems [92], role-based access control [73], trust management [16], delegation logic [57], and soft-

ware repositories [20–23, 71]. In the context of software repositories, delegations are specifically used to distribute permissions to sign packages across different administrators and developers. If A can sign a package K, then A can delegate this permission to B so that B can sign K on behalf of A. The delegation is an indirect package signature, where B "speaks for" [52] A about K.

Although the `projects` role may sign packages because it is the root of trust for all packages, in Figure 1 the `projects` role has instead *delegated* the Django project (or the package path `Django-*`) to the public keys belonging to the developer Alice. Similarly, the Scapy project has been delegated to Sue. A delegation is simply a trusted map of which developer keys are responsible for signing which projects (or sets of packages). Based on this delegation, users would trust only Alice's signature on a Django package. Developers can further delegate entrusted packages to other developers. In this case, Alice has delegated some packages (any package matching the path `Django-*.tar.gz`) to the developer Bob. Thus, Bob speaks for Alice for only the `Django-*.tar.gz` packages, whereas Alice's signature on `Django-1.7.1.exe` (not shown) would be trusted instead of Bob's.

3 Threats and Threat Model

There are many risks that users of software repositories face. Attackers can interject traffic by proxy interception attacks [43], target weaknesses in TLS [37, 85], set up a malicious mirror [21, 79], exploit weaknesses in the network infrastructure [19, 81], compromise signing keys due to weaknesses [45, 84], or steal keys outright by exploiting a security vulnerability [25]. Furthermore, attackers have proven adept at compromising the repository or signing infrastructure of many companies. This leads us to consider a threat model where a compromise of at least some part of the system occurs.

3.1 Threat Model

We assume that an attacker can:

1. Compromise a running repository and / or any keys stored on the repository, including those situations where the key itself is unknown (e.g., due to hardware protection) but where the attacker is nevertheless able to sign malicious packages using the key [67].

2. Respond to user requests. This can be done either by acting as a man-in-the-middle, or compromising the repository or one of its mirrors.

An attack will be successful if the attacker can change the contents of a package that a user installs (e.g., to insert a backdoor [27, 41, 43, 61, 62, 67]). Existing software update systems protect against a wide array of other attacks such as replay and mix-and-match attacks [20–23, 71]. We protect against those attacks by leveraging the role and delegation layout from these prior works. Thus, those types of attacks are only briefly discussed in this paper, so that we may focus on key compromise resilience while allowing online registration of projects.

We assume that projects have trustworthy developers and that these developers, who store their keys external to the community repository, take measures to secure them. If a key corresponding to a project is compromised, we consider a community repository's security to be effective if it limits the impact of an attack to the project whose key has been compromised.

4 Analysis of Current Systems

In this section, we examine four security approaches that are used on repositories. These techniques allow administrators of community repositories to sign packages — either themselves or by delegating packages to their respective project developers. These security models are illustrated in Figure 2. These models are discussed in turn in the following subsections.

4.1 Existing Security Models

(a) Repositories sign with online keys

In community repositories such as PyPI, RubyGems and npm, all packages are signed only by repositories with online keys (Figure 2(a)). A repository may sign packages with a transport mechanism such as TLS or CUP [42]. These private keys are kept online because community repositories must publish new projects and packages as soon as possible. Unfortunately, because the key is online, a compromise of the repository would instantly render all packages vulnerable. For example, the npm community repository reported that a programming bug not only leaked its TLS keys, but also allowed attackers to remotely rewrite packages [90]. Since developers do not sign their packages in this security model, users who subsequently request packages that have been tampered with trust the repository's package signatures without question. This is because the transport mechanism is useful only for establishing the identity of the repository, but not the authenticity of the packages themselves as belonging to their respective developers.

(b) Developers sign with offline keys

Some community repositories, including PyPI and RubyGems, permit developers to sign their packages with offline GPG [83] or RSA keys before uploading them to the repository (Figure 2(b)). Unlike the previous security model, signatures are used to verify the authenticity of packages, not to authenticate the repository's identity. In this model, users must discover the

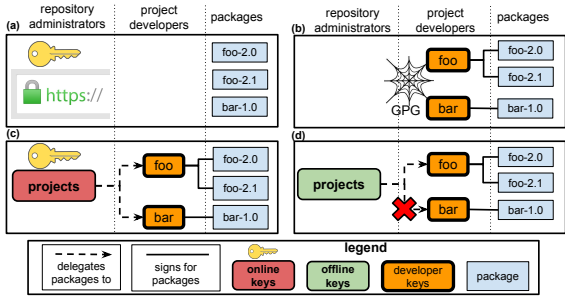

Figure 2: Existing security models for community repositories.

correct key for a developer from an out-of-band channel and then use this to verify packages.

One substantial problem with this model is that finding and verifying developer keys remains a manual process, with the burden placed on both developers and users. Finding true developer keys can be tricky, especially with attackers distributing fake keys, as was the case with the Tor project [24]. The repository is compromise-resilient only inasmuch as users have found and verified the correct developer keys. While PyPI and RubyGems support this model, only 4% of PyPI projects even list a signature. Moreover, in a month long trace of package requests to PyPI, only 0.07% of users downloaded these signatures for verification. If signatures are not used, then attackers who compromise the repository may modify any package in whatever manner they choose.

Thus, repository administrators across diverse community repositories are seeking a better solution [33, 47, 91]. To quote the RubyGems security guide [69]: "The goal is to improve (or replace) the signing system so that it is easy for authors and transparent for users."

(c) Repositories delegate to projects with online keys

In this security model (Figure 2(c)), the `projects` role for a delegation framework like TUF [71] is signed with an online key. In order to solve the problem of which developer keys map to which packages, repositories will *delegate* a project (its set of packages) to the public keys of the developers of that project. For example, PyPI will delegate *all* packages of the Django project (matching, say, the package path `Django-*`) to the public key of the lead developer of the project who, in turn, may delegate the Django packages to other developers. Since the `projects` role key is online, a new project can be immediately registered by the repository, through a new delegation to a project.

This model does not build compromise-resilient community repositories precisely because the `projects` role can be compromised by an attacker. (This is true despite the fact that developers sign their respective packages with offline keys.) This is because the keys for the `projects` role are kept online. Thus, once an attacker has compromised a repository, he (or she) is free to rewrite delegations using the online private keys. Then, an attacker can have the `projects` role delegate trust for the `Django-*` packages to a key that the attacker controls. As such, the attacker could deceive users into installing malicious packages that did not originate from the project's developers.

(d) Administrators delegate to projects with offline keys

Unlike the previous TUF security model, which delegates trust using an online key, administrators could alternatively choose to delegate using offline keys (Figure 2(d)). This means that the `projects` role key (kept offline) delegates projects to developer keys. Therefore, this model does indeed build compromise-resilient repositories because attackers cannot rewrite delegations (and thus packages) after a repository compromise. The attacker's capabilities are limited to preventing clients from seeing new packages in a timely manner (freeze attack) or providing new package updates out of order (mix-and-match attack) [71]. This model is used by traditional repositories, including LEAP [53]. Unfortunately, this model is impractical to use in community repositories because new projects, which are created dozens of times a day, cannot be registered without an administrator using an offline key.

5 Diplomat: Architecture and Delegations

This section describes the architecture of Diplomat, a security system designed to allow community repositories to have both compromise-resilience and immediate project registration. It begins with a high-level overview that explains the roles and use of delegations within Diplomat (Section 5.1). Following this, we present two problems that a delegation-based system would face when used on community repositories (Section 5.2). Diplomat addresses these problems using two types of delegations: prioritized delegations (Section 5.3) and terminating delegations (Section 5.4). In the next section, (Section 6) we will demonstrate how to use these delegations to provide compromise-resilience so that even if online keys for project registration are stolen, projects that were previously registered (with offline keys) are not at risk.

5.1 Roles and Delegations in Diplomat

Much like our earlier work on TUF [71], Diplomat separates trust between different parties using the four top-level roles shown in Figure 3: `root`, `timestamp`, `release`, and `projects`. Each role produces metadata that fulfills a specific purpose. The `root` role specifies the public keys of the top-level roles (including its own) and can revoke the other top-level role keys, if needed. The `release` role indicates the latest version numbers

Roles and Responsibilities	
root	The root role is the locus of trust. It indicates which keys are authorized for the projects, release, and timestamp roles. It also lists the keys for the root role itself.
projects	The projects role is trusted to validate packages. Often, the projects role will delegate trust for a project to the responsible developers.
release	The release role indicates the latest versions of all metadata on the repository. This prevents a user from later being deceived into installing an outdated package.
timestamp	The timestamp role is responsible for indicating if the repository contents have changed. This role will often be performed by external parties, such as mirrors.

Figure 3: The top-level roles used within Diplomat.

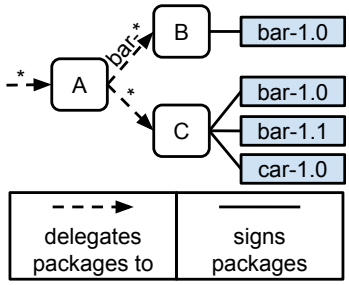

Figure 4: An example of ambiguous delegations. The label on a delegation specifies what packages to delegate.

of all Diplomat metadata (other than `timestamp`) that is available on the repository. The `timestamp` role references the latest `release` role metadata and will signify the last time the contents of the repository have changed. The `projects` role lists the available projects and either provides cryptographic hashes of packages or delegates trust to keys that provide those hashes. Each top-level role is only trusted for its assigned responsibilities; this minimizes the impact of a compromised role.

Our focus in this work is on the `projects` role (and the delegations it makes to non-top-level roles). Hence, details about any top-level role other than the `projects` role are only discussed as needed. Documentation is available that provides a more holistic discussion about the roles and their use [49, 50, 71].

The `projects` role is the root of trust for all packages on the repository; if a user wishes to download (and install) some package, he or she must first download and verify the latest `projects` role metadata. The user has the keys for this role because these public keys are contained within the `root` metadata file. The top-level projects role may delegate to other developers or vendors, which may also then delegate to others. A client can validate a package by following the chain of delegations until they find a trusted developer's metadata that contains the cryptographic hash of the package.

5.2 Problems With Delegation Ambiguity

Security problems can occur when a party cannot effectively control how much trust they place in a party when performing a delegation. To illustrate the problem, Figure 4 provides an example we will use throughout this section. In this example, A is the root of trust for packages. A delegates trust in any package with a name that matches the package path `bar-*` to B, and all packages (including `bar` packages) to C. This example illustrates two problems.

The Ordering Problem. Suppose that A has delegated `bar` to B and all packages to C. If B and C provide different cryptographic hashes for `bar-1.0`, which hash should be trusted?

Note that there is no "correct" resolution to this question, because A's intent is not clear. In some cases, the more specific delegation of `bar-*` to B should be trusted over the more general delegation of `*` to C. However, in other cases the reverse should be true. (In fact, the maximum security model in Section 6.1 has a general delegation which is prioritized over a specific delegation, which in turn is prioritized over another general delegation.) A solution must allow a party to express the intended order in which to resolve delegations.

The Failover Problem. Suppose that A wants B to be the only trusted party for `bar` packages, but C can be trusted with any other package. How can this be expressed?

There are clearly cases where failover is desirable (e.g., allowing a second developer to sign a package if the first does not) and those where it is not desirable (e.g., the ability for C to provide `bar-1.1`, if B is meant to be the only source of `bar` packages). A solution must enable the delegator to specify their intended behavior.

5.3 Prioritized Delegations

Diplomat uses *prioritized delegations* to order delegations between different parties and address the ordering problem. The key concept is to prioritize delegations based upon the order they occur in the metadata file. (This is similar to the manner in which firewall rules are processed in the order they are listed [60].) By exploiting the order in which delegations are listed, then the first delegation will be used before the second delegation, and so forth. For example, if B is listed before C in Figure 4, then the user would trust B over C for the `bar-1.0` package.

In case a role both delegates and signs a package, then the role's package signature takes precedence over its delegations. So if A signed the `bar-1.0` package, then A would be trusted for the package despite its delegation of the package to both B and C.

5.4 Terminating Delegations

Diplomat uses the concept of a *terminating delegation* to address the failover problem by halting the processing of delegations at a specified point. Terminating delega-

tions instruct the client not to consider future trust statements that match the delegation's pattern. This stops the delegation processing once this delegation (and its descendants) have been processed. (Handling this case is conceptually similar to the use of the cut operator in Prolog [18] to stop computation, except that Diplomat uses this technique for security instead of efficiency.) A terminating delegation for a package causes any further statements about a package that are not made by the delegated party or its descendants to be ignored.

5.5 Processing Delegations

The algorithm for resolving delegations through the application of prioritized delegations involves a pre-order, depth-first search of the `projects` metadata. This algorithm is used on a client device when a user instructs the package manager to install a package.

To install a package, a recursive algorithm begins at the `projects` role and searches for the package of interest. First, all of the hashes in the metadata file provided by the `projects` role are checked to see if the requested package is listed. This is the "pre-order" check to see if the current party has information about the desired package. Following this, delegations are examined in their order of priority (i.e., the order they are listed). If a delegation selects a portion of the namespace to delegate (e.g., `bar-*`), then the algorithm ignores this delegation if the pattern does not match (e.g., if the request was for `foo-1.0`). For any matching delegation, the algorithm will (in order) recursively search for the package of interest. It does so by repeating the preceding steps on the highest priority delegatees (in order). If any of the delegations is a terminating delegation, then the algorithm is terminated at that point, even if this terminating delegation does not return with an answer, preventing further delegations from being considered.

6 Diplomat Security Models In Practice

This section describes how to set up Diplomat metadata to provide real-time project registration and compromise-resilience. To exemplify how Diplomat is used in practice, we describe two security models that have been standardized for use within the Python community: the maximum security model (Section 6.1) [50] and the legacy security model (Section 6.2) [49]. After describing these two models, we elaborate on other usability aspects of Diplomat, such as handling key compromises, setting up roles, and maintenance in Section 6.3.

The legacy and maximum security models provide different trade-offs for the security and the availability of packages for projects we call *rarely updated*. A rarely updated project is one for which its developers have not provided a signing key, often because the package is not actively maintained. Nevertheless, its packages may be actively downloaded by users. An example is the BeautifulSoup project on PyPI, which last released package version 3.2.1 on February 16th, 2012, but nonetheless was downloaded more than a hundred thousand times in January 2016. The maximum security model uses an offline key to sign these projects. If an update is made, users will not receive it until the repository administrators sign the package with an offline key. In contrast, the legacy security model (Section 6.2) handles rarely updated projects by signing them with the online `unclaimed-projects` role. Due to the fact that online keys are used, developers can immediately update unclaimed projects. However, this comes at the cost of leaving their users vulnerable in the event of a repository compromise. Thus, the maximum model provides higher security but delayed availability of new packages for rarely updated projects, whereas the legacy model provides exactly the opposite trade-off. The security analysis of these models is available in our technical report [51].

6.1 Maximum Security Model

The maximum security model [50] (Figure 5) aims to reduce the risk to users if the repository is compromised by an attacker at the cost of making users wait before retrieving a new package for a rarely updated project. The top-level `projects` role of the maximum security model delegates to three other roles. The first and highest priority delegation, `claimed-projects`, is assigned to projects who have developers sign their own project metadata with their own offline key. The next highest priority delegation, `rarely-updated-projects`, requires repository administrators to delegate these projects with a terminating delegation, and to sign for these packages with an offline key. Finally, the lowest priority delegation, `new-projects`, is targeted to new projects, which are signed by an online key. If an attacker compromises the repository, they can change the metadata that indicates which key should be trusted for new projects (and thus can forge packages for those projects), but due to the higher-priority, terminating delegations to existing projects, whether rarely updated or not, cannot modify those packages without being detected.

The highest priority delegation issued by the `projects` role is to the `claimed-projects` role. The `claimed-projects` role signs a terminating delegation of all packages of a project (such as `foo` or `flibble`) to the public keys of its developers. Projects may choose to delegate trust to developers' public keys, and either a project or a developer will sign and upload metadata about their packages. The use of a terminating delegation ensures that if a user attempts to verify a `foo` package, then the user would only search for the package among its developers. Most importantly, the `claimed-projects` role signs its delegations

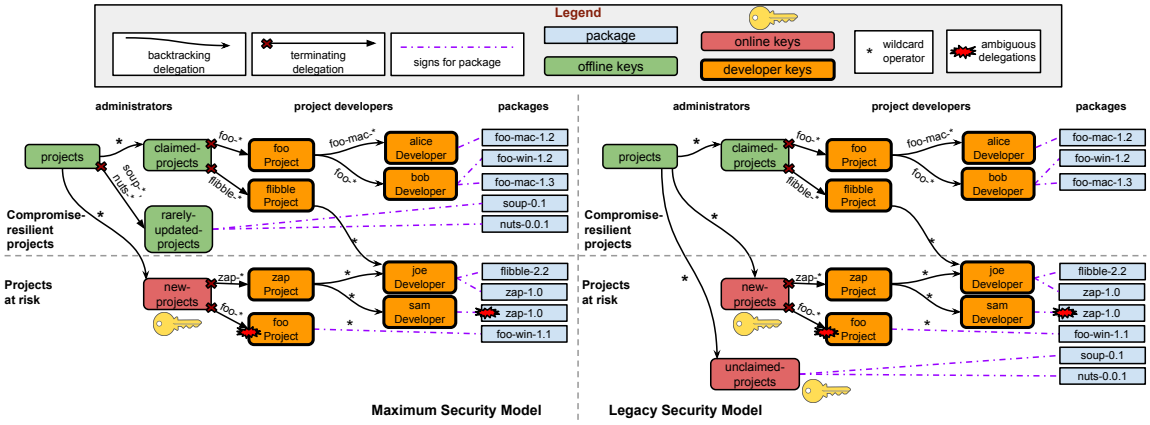

Figure 5: Maximum and legacy security models for community repositories. The red symbol indicates delegations that are not used due to earlier trust statements. Delegations that are higher on the figure (toward the top of the page) have higher priority.

to projects with *offline* keys so that attackers cannot tamper with the packages of these projects after a repository compromise (without also compromising the private keys used by claimed project developers). However, since the delegation from the `projects` role to the `claimed-projects` role allows backtracking (i.e., it is not a terminating delegation), any requests for projects unknown to the `claimed-projects` role will not be terminated at this role, and will instead continue with the `rarely-updated-projects` role.

The second-highest priority delegation pertains to the `rarely-updated-projects` role. This role directly signs, with *offline* keys, all packages of rarely updated projects. Since the key used is offline, packages cannot be signed by this role without an action by the repository administrators. This delays the release of new packages of rarely updated projects.

The delegation from the `projects` role to the `rarely-updated-projects` role is a terminating one. Furthermore, the `rarely-updated-projects` delegation specifies only the package paths of rarely updated projects (such as `soup-*` and `nuts-*` in Figure 5). Because of this, no backtracking is performed to search elsewhere for the package signatures of a project already delegated to this role.

Finally, the `new-projects` role is able to assign keys to package names that were not already defined. This role is served by an online key that delegates trust to newly created projects. However, since the role has an online key, there is a substantial risk of compromise. By assigning this role the lowest priority (and using prioritized, terminating delegations for claimed and rarely updated projects), an attacker will be able to only impact newly-created projects if the repository is compromised. For example, in Figure 5, the `new-projects` role's second delegation of `foo` is ignored due to the first terminating delegation of `foo` having a higher priority delegation via the `claimed-projects` role.

6.2 Legacy Security Model

The legacy security model [49] (Figure 5) is very similar to the maximum security model, but differs in the way that it handles rarely updated packages. This model allows new packages for rarely updated projects to be available immediately, while still providing security benefits to the users of claimed projects.

Like the maximum security model, the legacy security model includes the `claimed-projects` and `new-projects` roles. However, in the legacy security model, the repository uses an online key to sign for *unclaimed* projects. Like rarely updated projects, unclaimed projects are also signed by the repository instead of developers, but with the `unclaimed-projects` role that uses online keys. The `unclaimed-projects` role has the lowest priority delegation and, since the key is online, all projects signed with this key are at risk in the event of a compromise. Prioritized and terminating delegations of claimed projects signed by the `claimed-projects` role ensure that, even when the repository is compromised, packages of claimed projects are not at risk. Thus, all packages of unclaimed projects—unlike rarely updated projects—are available immediately, but vulnerable in case of a repository compromise (just like new projects).

The legacy model is drawn from our integration and deployment experience with the Python and Docker community repositories. Both repositories wanted to allow the repository to sign packages on behalf of developers who did not wish to do so. However, since its key is stored offline, using the `rarely-updated-projects` role would prevent administrators from quickly releasing new packages from these developers. The `unclaimed-projects` role permits the repository to im-

mediately sign packages on behalf of these developers.

Diplomat enables a repository to smoothly transition from the legacy to the maximum security model. The repository administrators can first have the `projects` role delegate to both the `rarely-updated-projects` and `unclaimed-projects` roles. The administrators can then move projects from the `unclaimed-projects` to the `rarely-updated-projects` role (Section 7.2.2), and / or require developers to register a project key to upload a new package. Either way, project developers will be incentivized to transition out of the `unclaimed-projects` role over time (using policies explored in Section 7), improving security.

Docker Hub uses a similar security model for its deployment of Diplomat. As of February 2016, Docker was signing the most popular projects, such as Ubuntu (a policy we explore in Section 7.2.1). Docker plans to explore options such as incentivizing project developers to sign their packages (by visually distinguishing or showing signed packages first in search results on Docker Hub), or requiring developers to sign packages in order to upload a new one (a policy we explore in Section 7.2.3).

6.3 Using Diplomat

Regardless of whether a repository uses the legacy or maximum security model, Diplomat requires essentially the same actions by users, developers and administrators.

Users. End users that install software through a package manager that uses Diplomat do not need to perform any actions and see no difference in their package manager's behavior. This is because Diplomat downloads and verifies its metadata before the package manager is allowed to install a package. The delegation structure in Diplomat manages keys on behalf of the user, which avoids the issues involved with locating and downloading appropriate project keys (e.g., the model in Figure 2(b)). The only situation where the user will be aware of Diplomat's existence is when a repository was compromised and it produces a message that notifies that signatures on data provided by the repository do not match.

Developers. To use Diplomat to protect a project, a developer must create a public / private key pair and upload the public key to their community repository. The repository will associate that key with the project first through the `new-projects` role and later through the `claimed-projects` role. If the project's leaders elect to do so, they may further delegate trust to different members of the project, who may also sign packages so that the project key need not be shared. Whenever a package is released, a developer must also generate a piece of signed Diplomat metadata, the format of which is in our standards document [49], that provides the cryptographic hash of the package. The actions needed to create or update this metadata can be added to the project's packaging scripts so that it is performed automatically when a new package is built. Diplomat provides a set of command-line tools [88] that helps developers to perform and automate those actions.

If the project key (i.e., the key that is delegated to directly by the `claimed-projects` or `new-projects` role) is compromised, the repository administrator will need to perform an action (discussed below) before trust in this key is revoked. However, if an individual developer key is compromised, the project can simply sign and upload a new piece of metadata that changes the key or removes that delegation. This action does not involve repository administrators.

Repository Administrators. Most of the work involved with using Diplomat comes from the initial setup. Repository administrators need to generate the keys for the roles and set up the initial delegations in their metadata. Offline keys should be stored in one or more devices that are not network connected and high value roles should require signatures from multiple keys. (We discuss procedures for this in more detail in the standards documents [49, 50].) The repository software needs to be modified so that Diplomat metadata is generated and updated whenever projects are registered or packages are uploaded. Diplomat provides administrators with command-line tools and APIs [89] that automate these actions and make it easy to integrate them into an existing repository.

Periodically (e.g., every few weeks), administrators will perform a maintenance operation on the repository to help it remain resilient to a key compromise. The administrators should append the `new-projects` role metadata to the `claimed-projects` role metadata and sign the resulting metadata with the `claimed-projects` key. If the `rarely-updated-projects` role exists, then newly uploaded packages that are not signed by their developers should be added. Revoked project keys, which are discussed below, are also replaced. Once this updated metadata file is uploaded, this makes it so that an attacker who compromises the repository cannot replace the key for any projects included before that point. Administrators will also calculate the cryptographic hash of every package on the repository and store this data on an offline system. This allows administrators to have a known-good hash of each package to detect and recover from a repository compromise.

Securely revoking a project key. When an authorized party wants to revoke trust in a project key, they notify the repository administrators and undergo an identity verification procedure [6]. (The exact procedure depends on the deployment and is out of the scope of this paper.) Once this is done, the administrators will write the new project key into a revoked role metadata file (not retrieved by users). When the maintenance operation is performed to generate the new `claimed-projects` file,

the revoked keys are replaced. Administrators may publish revoked project keys to Twitter as both a notification service and as a way of having a public log of project keys that will change in the next maintenance operation.

Securely recovering from a repository compromise. When a repository compromise has been detected, the integrity of three types of information must be validated. First, the keys for the `new-projects` and `unclaimed-projects` roles of the repository need to be revoked because they may have been compromised (i.e., their online keys have been compromised). The metadata for these roles must be discarded or returned to a known-good state. These keys can be revoked by having the offline `projects` role key sign new role metadata that delegates to a new key.

Second, the role metadata of the repository may have been changed. Metadata signed by the top-level `timestamp` and `release` roles may have been changed, enabling the attacker to launch mix-and-match and freeze attacks [20–23]. These keys should be revoked by the top-level `root` role, as is discussed in our prior work [71].

Third, the packages themselves may have been tampered with. Packages that existed the last time the `claimed-projects` and `rarely-updated-projects` role files were signed, can be verified using the stored hash information. Also, new packages that are signed by developers with the `claimed-projects` role may be safely retained. However, any package signed by developers using the `new-projects` or `unclaimed-projects` role should be discarded.

7 Evaluation

In order to better understand to what extent Diplomat makes community repositories compromise-resilient, we investigated the following questions:

- Does using the security models in Diplomat improve users' security in the event of a repository compromise? Is Diplomat better than existing solutions like TLS or GPG signatures? (Section 7.1)
- Suppose that the legacy security model is adopted for usability reasons. If the goal is to maximize security, what strategy should be used to get projects to sign packages? For example:
  - How effective is it to target the most popular projects? (Section 7.2.1)
  - What sort of benefit would there be from the repository signing rarely updated projects? Which projects should be considered rarely updated? (Section 7.2.2)
  - What is the effect of requiring developers to claim projects when uploading a package? (Section 7.2.3)
  - Is there an effective way to combine these strategies? (Section 7.3)

Quantitative data used to answer these questions was generated using anonymized request logs from PyPI from March 21st to April 19th, 2014. For the purposes of our analysis, we consider a user to be *vulnerable* if the user downloads at least one package that an attacker who compromised the repository (and thus all online keys) could have tampered with. Thus, mapping requests to user devices is important for our analysis. We sanitized the log to remove situations where a single IP address had diverse agent strings (likely multiple systems behind a NAT), or requested the same package more than once (likely a script), or requested more than 100 packages (likely a mirror). Sanitizing the request log reduced the absolute number of IPs in our dataset by about one third to 398,983 users, but provides a data set where each IP very likely corresponds to a unique client. This allows us to take a set of vulnerable packages and understand roughly what percent of clients would request at least one package in that set (and thus would be at risk).

For the purpose of our analysis, we assume that attackers have compromised PyPI on March 21st, at the beginning of the anonymized request log. Furthermore, when analyzing the maximum security model, we assume that all projects that existed before March 21st are delegated by the `claimed-projects` role, and that all projects created afterward during the compromise are delegated by the `new-projects` role.

7.1 Security of Diplomat vs TLS and GPG

We first perform a comparative analysis of the amount of risk placed on users if an attacker compromises (a) a repository protected with TLS, (b) GPG, (c) Diplomat's maximum security model, and (d) Diplomat's legacy security model. This analysis aims to find how effective these solutions would be in practice.

Figure 6 compares the effectiveness of TLS and GPG (the top-most line), and the maximum security model (very near the x-axis). In the case of TLS, the repository is trusted to indicate which packages are valid. Thus, if the repository is compromised, every user is vulnerable because users trust packages from the repository.

Even if GPG is used in conjunction with TLS, the security is not improved. So few developers sign packages and so few users download signatures that there was not a single user who downloaded only GPG-signed packages and their signatures.

Diplomat's maximum security model is not perfect, but it does protect 99.33% of users even if the repository compromise is not detected over the full month's trace. This is because most users only download packages from projects that existed before the start of the trace and those packages are not vulnerable. Users who are vulnerable

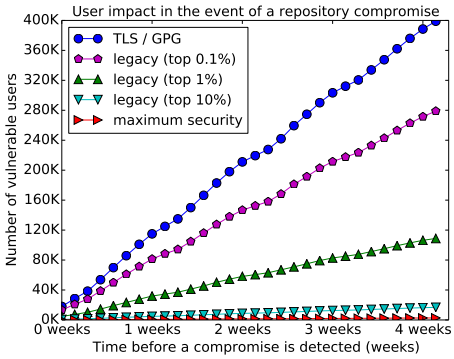

Figure 6: The cumulative number of compromised users over the month when popular projects are signed by developers (lower is better).

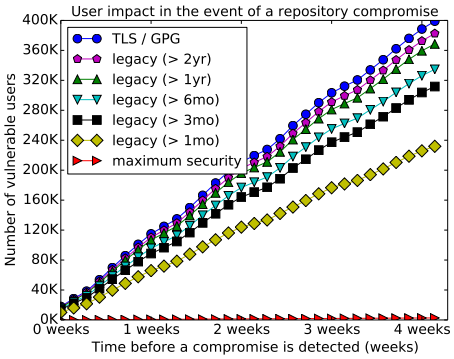

Figure 7: The cumulative number of compromised users over the month when the `rarely-updated-projects` role signed projects that were last updated before the specified time period.

are those who download a project that was registered during the compromise. This can happen when a new project rapidly becomes popular — often because of the "Slashdot effect" due to promotion on a news site.

7.2 Adoption Strategies in the Legacy Security Model

The compromise-resilience offered by the legacy security model can range from the same as TLS and GPG — none, if no project signs its packages — to as good as the maximum security model, if administrators delegate with offline keys all but new projects to their respective developers. In the rest of this subsection, we explore how the compromise-resilience of the legacy security model differs when different types of projects adopt Diplomat.

7.2.1 Targeting popular projects

We first evaluated the impact of requiring developers of the most popular projects to claim their packages (Figure 6). Increasing the number of popular packages that are signed by developers dramatically increases security. If the most popular 1% of projects are signed by developers (406 projects), then 73% of users are protected. If the top 10% of projects sign their project, then 96% of users are protected. This shows that users overwhelmingly download only popular projects and so focusing on their protection is highly effective.

7.2.2 Only signing rarely updated projects

We examined the security benefits of the repository using an offline key to sign rarely updated projects, because this has very little usability impact until the project is next updated.

As Figure 7 shows, the security benefit of signing rarely updated projects is small. This is because many popular projects are updated frequently. Even if projects that have not updated merely in the last month are considered rarely updated, only 167,097 (42%) of users would be protected if the repository were compromised.

Creating a new package for a rarely updated project means that users will not see the update until repository administrators sign the package with the `rarely-updated-projects` role. This is a major usability problem and so the rate of projects that are considered rarely updated must be very low. To estimate this, we examined the distribution of the maximum time difference between consecutive package updates for all projects (not shown). Our analysis shows that 12% of all packages had a gap of at least a year between updates, but only 4% had a gap of at least two years. We feel that two years is the most aggressive setting for rarely updated projects that is likely to be considered acceptable by the Python community. However, due to the high rate of false positives and low number of users protected, on its own, this is not an effective strategy.

7.2.3 Requiring projects to sign to upload a package

We considered a strategy wherein PyPI would require projects to sign packages in order to upload a new package. (We consider this from a security perspective and ignore the community's response to such a policy.) Figure 8 shows the relative impact on users to be dependent on how long the policy has been in place. For example, the magenta line ("legacy (last 3mo)") shows that if developers that updated a package within the last three months signed that package, 247,969 users (62%) were vulnerable.

The usefulness of requiring a signature to upload a package tails off rather sharply. Somewhat surprisingly, 23% of users (90,091) were vulnerable even if all developers that uploaded a package in the last two years signed it (27,235 projects). This means that many popular projects have not been recently updated. (Given the observation in the previous subsection, many popular packages are updated frequently, yet many are not.) In comparison, this is about as effective as signing the most popular 1% of projects, despite only requiring an

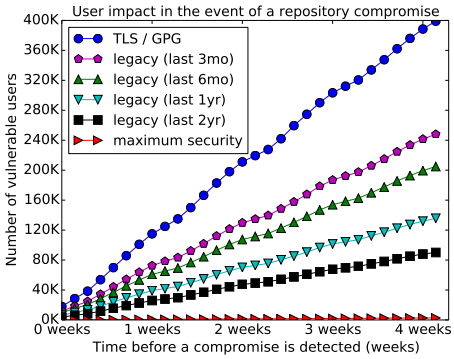

Figure 8: The cumulative number of compromised users over the month when projects were gradually signed by developers over time using the legacy security model.

action by 406 projects. Thus, requiring projects to sign when uploading is not an effective strategy when used in isolation.

7.3 Summary: Recommended strategy

To summarize, pushing for adoption by the most popular 1% projects is critical to securing users. Other strategies, such as signing for rarely updated projects and pushing projects to sign when uploading a package, will further help security. While each strategy may be relatively ineffective on its own, combining all of these strategies can have minimal usability impact while greatly increasing the security of PyPI users until the maximum security model is adopted. The details of this analysis, as well as our implementation of Diplomat, are available in our technical report [51]. The Diplomat source code and standards documents are freely available at `https://theupdateframework.github.io/`.

8 Related Work

Role-Based Access Control. Diplomat uses a role-based access control (RBAC) [72, 73] model. RBAC is a collection of security models where permissions are associated with roles. A user may belong to one or more role and thus possess the permissions those roles provide. Although delegations in RBAC are well studied [8–10], ambiguous delegations are not generally studied because users, permissions, and roles are usually implicitly assumed not to conflict. Schaad [74] used Prolog to detect conflicts in separation of duties between roles, but discussed no resolution mechanism that is applicable beyond RBAC. Stork [20] and OrBAC [14, 29] support delegation models that allow the simultaneous permission and prohibition of a privilege, and so could have the same ambiguity issues as this work. These systems solve the problem by specifying *priorities* with every permission and prohibition. However, unlike Diplomat, these systems assume that the metadata a party sees cannot be controlled by the attacker. An attacker that compromises a community repository can choose to omit, add, or, in the case of online keys, alter metadata, which is not handled by these schemes.

Trust Management. Although GPG [83] and X.509 [26] are useful for finding public keys and telling whether keys have been revoked, they cannot answer the question [17]: "Is request r authorized by policy P and credential set C?" In a seminal work [16], Blaze et al. defined *trust management* and introduced PolicyMaker, a general trust management engine to enforce security policies for diverse applications. PolicyMaker separates secure key distribution (as solved by GPG or X.509) from distributed authorization. PolicyMaker featured security principles shared by Diplomat such as k-of-n thresholds for authorization, deferred trust (delegations), local policies, and key revocation. Later, KeyNote [15] more directly supported public-key infrastructure-like applications with a simpler syntax and semantics at the expense of generality. Unlike Diplomat, PolicyMaker and KeyNote do not handle conflicting statements made by apparently equally trustworthy parties [57]. Like KeyNote, SPKI/SDSI [34] supports trust management for public key infrastructures (PKI). SPKI/SDSI chose a delegation model with boolean control; unlike Diplomat, a key holder can specify the inability to delegate further. SPKI/SDSI considered many security problems that are relevant to Diplomat: key revocation, the risks of online keys and increased key lifetimes, redundancy with a threshold of algorithms or keys, and applying redundancy to replace root keys. However, unlike Diplomat, SPKI/SDSI leaves the processing of delegations, including conflict resolution, to application developers.

Delegation Logic. Diplomat leverages ideas from prior work in logic-based distributed authorization. Much of Diplomat's functionality may be expressed in D2LP, an authorization language in delegation logic [44, 56–58]. D2LP extends early works on trust management and authorization in distributed systems [1, 52] by defining a non-monotonic logic that resolves conflicting conclusions in security policies. D2LP defines a more general notion of prioritized delegations than in Diplomat. For example, although both allow delegators to constrain delegations, D2LP allows delegators to specify arbitrary delegation depth, whereas delegations are always infinitely deep in Diplomat. Furthermore, D2LP allows for partial ordering of rules, but Diplomat requires all delegations to be totally ordered. This means that there will *always* be one trusted conclusion for a package's metadata, or none at all in case no administrator or developer has signed for the package.

Secure Software Updates. Problems with software update security were examined by Bellissimo et al. [13]

and Cappos et al. [21]. More recently, Knockel et al. [46] observed that man-in-the-middle attacks on third-party software continue to beleaguer open infrastructure.

The Stork package manager [20, 22, 23], whose security model is also used by popular Linux package managers, addresses a wide array of attacks that involve malicious mirrors. However, this security model assumes that the repository is trustworthy. TUF [71] is designed to securely handle situations where some or all of a repository is compromised. Diplomat leverages the techniques in TUF to protect against certain types of attacks, such as attacks that make valid but outdated packages appear current. However, as was discussed in Section 4, due to the need for online project registration TUF cannot protect a community repository against the most impactful attacks, such providing arbitrarily modified packages.

Revere [55] uses a self-organizing, peer-to-peer overlay network to deliver security updates. It is designed to maximize delivery speed, scalability, high dissemination assurance, and security. In Revere, each peer independently decides on trust relationships with other peers in the overlay network. Thus from a security standpoint, Revere functions somewhat like GPG. In contrast, Diplomat works with a central community repository and delegates trust to projects which do not host content themselves.

Meteor [11] is designed to secure smartphones against multi-market environments. Relevant to Diplomat is their assumption that updates can be malicious due to a compromise of developer keys. They propose independent databases of metadata (such as information about developers, or rating of application binaries by experts) that users consult to determine whether an application should be trusted. Baton [12] showed how smartphone application developers can securely transfer the signing authority of an application to a new developer key without requiring user intervention and a PKI.

Alhamed et al. [2, 3] studied a different approach to securing community repositories. They propose a volunteer community of independent testers who build binaries from releases, certify that binaries come from trusted sources, and attach warnings or even praises to binaries.

Secure Software Repository. The Secure Untrusted Data Repository (SUNDR) [54] addresses a different threat model from Diplomat. SUNDR assumes that the software repository itself cannot be trusted with storing packages. Therefore, each developer checks and signs the history of all file system operations. Developers compare histories with each other in order to ensure *fork consistency*, where developers can eventually detect equivocation. SUNDR requires developers and users to mount a SUNDR file system hosted on the repository, and use its protocol to verify the file system history.

9 Conclusion

This paper presents an architecture that uses prioritized and terminating delegations to secure community repositories. The architecture demonstrates that it is possible to have compromise-resilience in a community repository without sacrificing a defining feature: immediate project registration.

Our system, Diplomat, is flexible enough to enable community repositories to implement different security policies and gracefully transition between them. A community repository can begin with the legacy security model, which provides sufficiently strong protection, but does not require any action by developers. The maximum security model does require that developers sign their packages (or else, new packages cannot be immediately released); however, the security gains are substantial. Diplomat's maximum security model would protect over 99% of PyPI users, even if an attacker controlled the repository undetected for a month.

Acknowledgements

We thank our shepherd, Ramakrishna Kotla, as well as Jon Howell and the anonymous reviewers for their valuable comments. We would also like to thank Lois Anne DeLong and Linda Vigdor for their efforts on this paper, as well as the Docker, Flynn, Haskell, LEAP, OCaml, Python, Ruby, and Square communities for their collaboration. Our work on Diplomat was supported by U.S. National Science Foundation grants CNS-1345049 and CNS-0959138.

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] ALHAMED, K., SILAGHI, M. C., HUSSIEN, I., STANSIFER, R., AND YANG, Y. "Stacking the Deck" Attack on Software Updates: Solution by Distributed Recommendation of Testers. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on* (2013), vol. 2, pp. 293–300.

[3] ALHAMED, K., SILAGHI, M. C., HUSSIEN, I., AND YANG, Y. Security by Decentralized Certification of Automatic-Updates for Open Source Software controlled by Volunteers. In *Workshop on Decentralized Coordination* (2013).

[4] APACHE INFRASTRUCTURE TEAM. apache.org incident report for 8/28/2009. `https://blogs.apache.org/infra/entry/apache_org_downtime_report`, 2009.

[5] APACHE INFRASTRUCTURE TEAM. apache.org incident report for 04/09/2010. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`, 2010.

[6] ARCIERI, T. Let's figure out a way to start signing RubyGems. `http://tonyarcieri.com/lets-figure-out-a-way-to-start-signing-rubygems`, 2014.

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

[8] BARKA, E., AND SANDHU, R. Role-based delegation model/hierarchical roles (RBDM1). In *Computer Security Applications Conference, 2004. 20th Annual* (2004), IEEE, pp. 396–404.

[9] BARKA, E., AND SANDHU, R. Framework for agent-based role delegation. In *Communications, 2007. ICC'07. IEEE International Conference on* (2007), IEEE, pp. 1361–1367.

[10] BARKA, E., SANDHU, R., ET AL. A role-based delegation model and some extensions. In *23rd National Information Systems Security Conference* (2000), Citeseer, pp. 396–404.

[11] BARRERA, D., ENCK, W., AND VAN OORSCHOT, P. C. Meteor: Seeding a security-enhancing infrastructure for multi-market application ecosystems. *IEEE Mobile Security Technologies* (2012).

[12] BARRERA, D., MCCARNEY, D., CLARK, J., AND VAN OORSCHOT, P. C. Baton: Key Agility for Android without a Centralized Certificate Infrastructure. Tech. Rep. TR-13-03, School of Computer Science, Carleton University.

[13] BELLISSIMO, A., BURGESS, J., AND FU, K. Secure software updates: disappointments and new challenges. *Proceedings of USENIX Hot Topics in Security (HotSec)* (2006).

[14] BEN-GHORBEL-TALBI, M., CUPPENS, F., CUPPENS-BOULAHIA, N., AND BOUHOULA, A. A delegation model for extended RBAC. *International journal of information security 9*, 3 (2010), 209–236.

[15] BLAZE, M., FEIGENBAUM, J., AND KEROMYTIS, A. D. Keynote: Trust management for public-key infrastructures. In *Security Protocols* (1999), Springer, pp. 59–63.

[16] BLAZE, M., FEIGENBAUM, J., AND LACY, J. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (1996), IEEE, pp. 164–173.

[17] BLAZE, M., FEIGENBAUM, J., AND STRAUSS, M. *Financial Cryptography: Second International Conference, FC '98 Anguilla, British West Indies February 23–25, 1998 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, ch. Compliance checking in the PolicyMaker trust management system, pp. 254–274.

[18] BRATKO, I. *Prolog programming for artificial intelligence*. Pearson education, 2001.

[19] BROWN, M., AND DYNAMIC NETWORK SERVICES, INC. Pakistan hijacks YouTube. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/, 2008.

[20] 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).

[21] 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.

[22] CAPPOS, J., SAMUEL, J., BAKER, S., AND HARTMAN, J. H. Package management security. *University of Arizona Technical Report* (2008), 08–02.

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

[24] CLARK, E. [tor-talk] Another fake key for my email address. https://lists.torproject.org/pipermail/tor-talk/2014-March/032308.html, 2014.

[25] CLOUDFLARE, INC. Answering the Critical Question: Can You Get Private SSL Keys Using Heartbleed? https://blog.cloudflare.com/answering-the-critical-question-can-you-get-private-ssl-keys-using-heartbleed/, 2014.

[26] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *The Internet Society* (2008). https://tools.ietf.org/html/rfc5280.

[27] CORBET, J. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

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

[29] CUPPENS, F., CUPPENS-BOULAHIA, N., AND GHORBEL, M. B. High level conflict management strategies in advanced access control models. *Electronic Notes in Theoretical Computer Science 186* (2007), 3–26.

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

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

[32] DOCKER INC. Docker Hub. https://hub.docker.com/.

[33] EKLEKTIX, INC. Docker image "verification". https://lwn.net/Articles/628343/, 2015.

[34] ELLISON, C., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND YLONEN, T. RFC 2693: SPKI certificate theory. https://tools.ietf.org/html/rfc2693.

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

[36] GENTOO LINUX. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[37] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 38–49.

[38] GESBERT, L., AND MEHNERT, H. Signing the OPAM repository. http://opam.ocaml.org/blog/Signing-the-opam-repository/, 2015.

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

[40] GNU SAVANNAH. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[41] 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.

[42] GOOGLE, INC. Open Client Update Protocol. http://omaha.googlecode.com/svn/wiki/cup.html.

[43] GOSTEV, A. 'Gadget' in the middle: Flame malware spreading vector identified. https://www.securelist.com/en/blog/208193558/Gadget_in_the_middle_Flame_malware_spreading_vector_identified, 2012.

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

[45] INCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/.

[46] 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.

[47] KRAH, S. [Python-Dev] pip: cdecimal an externally hosted file and may be unreliable [sic]. https://mail.python.org/pipermail/python-dev/2014-May/134453.html, 2014.

[48] KUHN, B. M. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. https://savannah.gnu.org/forum/forum.php?forum_id=2752, 2003.

[49] 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.

[50] KUPPUSAMY, T. K., DIAZ, V., STUFFT, D., AND CAPPOS, J. PEP 480 – Surviving a Compromise of PyPI. https://www.python.org/dev/peps/pep-0480/, 2014.

[51] 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.

[52] 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.

[53] LEAP ENCRYPTION ACCESS PROJECT. New releases for a new year - LEAP. https://leap.se/en/2014/darkest-night, 2014.

[54] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 9–9.

[55] 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.

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

[57] 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.

[58] 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.

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

[60] MICROSOFT, INC. Order of Windows Firewall with Advanced Security Rules Evaluation. https://technet.microsoft.com/en-us/library/cc755191%28v=ws.10%29.aspx, 2009.

[61] 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.

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

[63] MÓNICA, D., AND DOCKER, INC. Introducing Docker Content Trust. https://blog.docker.com/2015/08/content-trust-docker-1-8/, 2015.

[64] PHILIPS, B. Evaluate The Update Framework. https://github.com/appc/spec/issues/211, 2015.

[65] PRIME DIRECTIVE, INC. Development - Flynn. https://flynn.io/docs/development, 2015.

[66] PYTHON SOFTWARE FOUNDATION. PyPI - the Python Package Index: Python Package Index. https://pypi.python.org/pypi.

[67] RED HAT, INC. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[68] RUBYGEMS.ORG. RubyGems.org — your community gem host. https://rubygems.org/.

[69] RUBYGEMS.ORG. Security. http://guides.rubygems.org/security/.

[70] RUBYGEMS.ORG. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[71] 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.

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

[73] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[74] SCHAAD, A. Detecting conflicts in a role-based delegation model. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual* (2001), IEEE, pp. 117–126.

[75] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 1. https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-1.html, 2013.

[76] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 2. https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-2.html, 2013.

[77] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 3. https://corner.squareup.com/2013/12/securing-rubygems-with-tuf-part-3.html, 2013.

[78] SLASHDOT MEDIA. About. http://sourceforge.net/about.

[79] SLASHDOT MEDIA. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

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

[81] STEWART, J. DNS cache poisoning–the next generation. http://www.secureworks.com/research/articles/dns-cache-poisoning, 2003.

[82] THE FREEBSD PROJECT. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.

[83] THE GNUPG PROJECT. The GNU Privacy Guard. https://gnupg.org/.

[84] THE MITRE CORPORATION. CVE 2008-0166. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166, 2008.

[85] THE MITRE CORPORATION. CVE 2014-0092. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0092, 2014.

[86] THE PHP GROUP. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.

[87] THE PHP GROUP. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.

[88] THE UPDATE FRAMEWORK. Developer Tools. https://github.com/theupdateframework/tuf/blob/develop/tuf/README-developer-tools.md.

[89] THE UPDATE FRAMEWORK. Repository Management. https://github.com/theupdateframework/tuf/blob/develop/tuf/README.md.

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

[91] WELL-TYPED LLP. Improving Hackage security. http://www.well-typed.com/blog/2015/04/improving-hackage-security/, 2015.

[92] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS) 12*, 1 (1994), 3–32.