Survivable Key Compromise in Software Update Systems

Justin Samuel*
UC Berkeley
Berkeley, California, USA
jsamuel@berkeley.edu

Nick Mathewson
The Tor Project
nickm@alum.mit.edu

Justin Cappos
University of Washington
Seattle, Washington, USA
justinc@cs.washington.edu

Roger Dingledine
The Tor Project
arma@mit.edu

ABSTRACT

Today's software update systems have little or no defense against key compromise. As a result, key compromises have put millions of software update clients at risk. Here we identify three classes of information whose authenticity and integrity are critical for secure software updates. Analyzing existing software update systems with our framework, we find their ability to communicate this information securely in the event of a key compromise to be weak or nonexistent. We also find that the security problems in current software update systems are compounded by inadequate trust revocation mechanisms. We identify core security principles that allow software update systems to survive key compromise. Using these ideas, we design and implement TUF, a software update framework that increases resilience to key compromise.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.0 [**Software Engineering**]: Protection Mechanisms

General Terms

Security, Design

Keywords

Software updates, authentication, delegation, key management, key compromise, revocation, threshold signatures

*Work done while at the University of Washington.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

1. INTRODUCTION

Software update systems span a wide range of uses and designs. This includes their use as package managers [4, 59, 60], library managers [16, 20, 44, 48], and application updaters [34, 36]. All of these software update systems play a critical role in computer security. They must discover, download, verify, and apply updates when security flaws are discovered in installed software. Crucial to the security of this process is that all downloaded files are ensured to be authentic. Though some systems do not authenticate updates [2, 7, 11], many do. In practice, authentication in these systems requires cryptographic signatures. The security of software update systems therefore relies heavily on the inability of attackers to obtain private keys that are trusted to provide updates.

Historically, software update systems have been designed such that the compromise of a single trusted key is fatal. With a compromised key, attackers who can respond to client requests can cause clients to install malicious software. In many systems, even if the developers learn of a key compromise, there is no secure means of key revocation. This risk is not theoretical: popular Linux distributions with tens of millions of users have suffered panics due to key compromise [23, 46], weak key generation is a continuing threat [12, 28, 58], and there are many known PKI vulnerabilities [30, 53]. Recently, attention has focused on the possibility of government-compelled certificate issuance, a tool that governments may use to tamp down on political dissidents [13, 52]. While that work highlighted the risk of such attacks to secure web browsing, their threat extends into software update systems, which commonly rely on PKI for transport layer security as well as code signing. Having compromised a key by any of these means, attackers can impersonate update servers using DNS cache poisoning [19], BGP prefix hijacking [42], posing as (or compromising) a legitimate content mirror [11], or any other method of intercepting client requests.

Ideally, software update systems would remain safe even when some of their keys are under the control of an attacker. This property of *survivability*—the ability for a system to function correctly while under attack or partial compromise—is important for building reliable systems. We believe a software update system cannot remain secure if an attacker who can respond to client requests has compromised *all* its keys. However, with the proper understanding of software update system vulnerabilities, we find updaters can be designed such that specific attacks requiring partial

key compromise are less likely to succeed. Additionally, the severity of many successful attacks can be decreased.

We begin by identifying the information that software update systems must authenticate in order to perform secure updates. This information is the content of updates, the availability (timeliness) of updates, and the consistency of information describing updates. Unfortunately, the keys that software update systems use to protect this information are vulnerable to compromise in many ways. We examine popular software update systems, all of which authenticate some data, and find no principled approach to securing this information in the event of key compromise.

We then define a set of design principles that can improve the resilience of software update systems to key compromise. We have applied these principles in our design of TUF, an open source software update framework. TUF is the next generation of the Thandy [33] design originally developed for secure updates of Tor [54]. The design of TUF uses responsibility separation and delegation, multi-signature trust with threshold signatures and multiple roles, and both implicit and explicit trust revocation. TUF can be integrated with both new and existing software update systems. We discuss our experiences integrating TUF with two dissimilar software update systems: a traditional application updater and Python's library management system. Based on our experiences, we discuss practical implications of our design and make recommendations for key threshold sizes, metadata expiration times, and appropriate use of automated signing.

Main contributions:

- We draw attention to the danger of relying on single signing keys and PKI[1] in software update systems. PKI is exceptionally high-risk in update systems as the software provider does not control the trust chain. We look at a variety of update systems and find partial or complete reliance on PKI in most of these systems.
- We identify three fundamental classes of information that software update systems must authenticate to ensure correct and timely updates. This information is the content of updates, the availability of updates, and the correct combination of updates.
- We show how role separation can be used in software update systems to reduce the impact of key compromise. Central to this separation is the observation that an individual key's manner of use and storage affects its likelihood of compromise. We demonstrate how responsibilities can be divided between roles whose keys have different likelihoods of compromise in order to reduce the overall risk to clients.
- We design and implement a software update system that uses multiple-signature trust and role separation without requiring SSL or other protocols that store keys on public-facing servers.

Paper organization: A threat model is given in Section 2. Software updates systems are described along with their vulnerabilities and information responsibilities in Section 3. Threats to key security are discussed in Section 4. The security properties of current software update systems are examined in Section 5. Design principles for securing software update systems are identified in Section 6. We describe the design of our framework and analyze its key compromise survivability in Section 7. We discuss our experiences and make recommendations for balancing security with usability in Section 8. We look at related work and conclude in Sections 9 and 10.

[1]We use the term *PKI* to refer to contemporary PKI based on third-party CAs. Our design includes what is, in fact, a public key infrastructure. To avoid ambiguity, we will not refer to it as a PKI.

2. THREAT MODEL

Our threat model assumes the following:

- Attackers can compromise at least one of a software update system's trusted keys.
- Attackers compromising multiple keys may do so at once or over a period of time.
- Attackers can respond to client requests.

We consider the attacker as successful if they can convince the updater to install (or leave installed) something other than the most up-to-date version of the software it is updating. If the attacker is preventing the installation of updates, they want the updater to not realize there is anything wrong.

We deem the ability for an attacker to have arbitrary files signed by a key to be functionally equivalent to a key compromise. To a software update system, the ability to sign arbitrary files is indistinguishable from possession of the key.

We assume that the internal processes used by project administrators to manage their codebase and software are secure. Though important for overall client security, these processes are out-of-scope for software update systems.

3. SOFTWARE UPDATE SYSTEMS

Software update systems find, download, and install the latest versions of software. They usually begin by making requests to servers that host the updates. The server responds with information describing available updates and the update system then decides which updates to download and install. Often, the servers from which updates are downloaded are not owned by the same organization that created the updates. These third-party repository mirrors may be provided by volunteers or may be owned by content delivery networks (CDNs).

Software update systems span a range of purposes. Package managers such as YUM [60] and APT [4] are responsible for most of the software installed on an operating system. Library managers such as Python's easy_install [20] are commonly provided with programming environments to install and update optional libraries. Application updaters are included with individual applications such as Firefox so that the application can update itself without needing, for example, a package manager, or manual intervention by the user.

3.1 Authentication

Software update systems must verify the authenticity of files they receive. These files may be the actual software updates, may describe available updates, or may provide other information that the updater uses to make download and trust decisions. Software update systems that authenticate none of this information are insecure and exploiting them has been well studied [2, 7].

Information	Vulnerability
Content of updates	Malicious software
Timeliness of updates	Freeze attack
Repository state	Metadata inconsistency

Table 1: Information responsibilities in software update systems and the corresponding vulnerabilities that exist when these responsibilities are compromised.

In this work, we are not concerned with systems that offer no security. Instead, we focus on systems designed for security whose architectures are nevertheless dangerously fragile with respect to key compromise. Among the systems that do authenticate the information they receive, the two primary methods of authentication are transport layer security and cryptographically signed files.

Many software update systems rely on transport layer security [34, 36]. These systems verify that they are retrieving trusted files by using an authenticated transport protocol (such as SSL/TLS), and verifying the public key certificate of the server from which they retrieve updates. The security of these systems is heavily based on the security of an underlying PKI. Trust is bootstrapped by the client having a set of trusted root CA certificates.

Other software update systems use signed data [4, 59, 60]. In these systems, clients know one or more keys that they trust to provide updates. When files are downloaded, the signatures on these files are checked.

A few systems use both transport layer security and signed files [59, 60]. Of these, some rely on PKI-based file signing. Thus means that despite the use of code signing, clients are still vulnerable to compromise as a result of PKI problems such as those we will discuss in Section 4.

It is common for software update systems to directly authenticate only a single metadata file through either a signature on the file or transport layer security. This metadata file includes cryptographic hashes of all other available metadata and update files. By comparing the hashes of the files downloaded over insecure channels with those listed in the initial signed or SSL-transferred metadata, updaters ensure the authenticity of these other files. This technique is especially useful to larger projects that rely on SSL, as SSL has a significant performance impact on servers providing updates for many clients.

3.2 Information Responsibilities

To address key compromise in software update systems, we must consider attacks specifically aimed at them. Various attacks on software update systems are described in [11]. Here we classify the underlying vulnerabilities and identify the information that software update systems must authenticate in order to remain secure. These information responsibilities and the related vulnerabilities are listed in Table 1 and described below.

3.2.1 Content of Updates

For an attacker to compromise clients through the update process, the easiest approach is to provide malicious software to a client downloading updates. To protect against installation of malicious updates, software update systems must be able to verify that all updates are authentic.

3.2.2 Timeliness of Updates

If an attacker can prevent a software update system from learning that updates are available, the attacker can also prevent the installation of security updates. As a result, attackers gain time to exploit known vulnerabilities in software installed on client systems. We call this a *freeze attack*.

To prevent freeze attacks, clients must know when updates are available. Though a client cannot stop an attacker from preventing updates altogether, a client should be able to detect when an attacker is replaying responses to the client's previous queries for updates. Thus, clients must be able to authenticate information that indicates the availability of updates.

3.2.3 Repository State

In a *metadata inconsistency attack*, a software update system is tricked into using a combination of metadata and update files that never existed together on the repository at the same time. Depending on the design of the software update system, the result of this attack varies. For example, a metadata inconsistency attack may allow an insecure version of a dependency to be installed [11].

We call the set of files available from a repository at a given time the *repository state*. To remain secure against metadata inconsistency attacks, software update systems need access to authenticated information that indicates whether any subset of downloaded files is from the same repository state.

4. THREATS TO KEY SECURITY

There are many ways signing keys and transport keys can be compromised or bypassed. Some of these categories of risk have been considered before in the context of key compromise [27].

4.1 PKI Vulnerabilities

Trust based on a chain of single keys is compromised if any key in the chain is compromised. Thus, the use of traditional PKI for trusting a key within an update system, whether that key is for signing files or for transport layer security, is problematic. For example, a single trusted CA certificate held by a malicious party puts all clients at risk.

Recently, the risk of rogue CA certificates was shown to be more than theoretical [53]. As many clients rely on certificates to indicate the location of a revocation server (OCSP responders or CRLs), rogue CA certificates may be difficult to revoke. This risk of rogue CA certificates is in addition to the inherent danger of compromise of any CA's private key.

With PKI, there is also the risk that one of the certificate authorities improperly issues certificates. A notable case of a fraudulently issued certificate is the issuing by VeriSign of a certificate in the name of Microsoft Corporation to an attacker claiming to be a Microsoft employee [15]. Incorrectly issued certificates are also fundamental to the *null prefix attack* discovered in 2009 [31]. This attack uses a certificate whose common name is formatted as the domain to impersonate, followed by a null byte, followed by a domain owned by the attacker. Many CAs would sign such a certificate and many SSL implementations would consider only the portion of the common name before the null byte. As a result, an

attacker could obtain a certificate that would be trusted for any domain.

Fraudulently issued certificates can also be the result of compelled certificate creation. In such an attack, a government agency compels a CA to issue a certificate in order to enable the agency to intercept secure communication [52].

Despite the risks of PKI-based authentication, many software update systems rely solely on PKI for authenticating updates. Examples of such systems include Firefox's built-in updater [22] as well as Google's Windows and Mac updaters [24, 25].

4.2 Key Theft

Private keys can be stolen either by an outsider gaining unauthorized access to a system storing the key or by an insider with access to the key.

The more easily accessible a key is, the easier it is to steal. In the case of transport keys, the private key will often exist unencrypted on disk and will always exist in memory unencrypted. If the server software is compromised, a private key in the process's memory will not be safe even when access control mechanisms such as SELinux are used. Even without obtaining the private key, an attacker who has compromised such a system can impersonate the repository and reply to requests from software update clients. The compromise of web servers is not uncommon. There have been many remotely exploitable vulnerabilities in web server software [3, 26] as well as in SSL communication libraries used by web servers [35, 43].

Even though signing keys are generally maintained more securely than transport keys, they can still be compromised if an attacker gains access to an internal server through vulnerabilities or compromised accounts. Two recent high-profile compromises of software signing systems are those of the popular Linux distributions Fedora and Red Hat.

In August 2008, the server that was used to sign official Fedora software was compromised [23]. This server had on it a passphrase-protected key. Fedora staff would type the passphrase on the system when software needed to be signed. If the attacker was able to obtain the key's passphrase either directly (for example, though keystroke logging) or through brute force, they would have been able to gain control of any Fedora client that performed an update or new software installation. Fedora distributed a new key to clients through an insecure means (using the potentially compromised key to sign a package to replace the key). Fedora's website currently states that they intend to consider better methods for key migration in the future [21].

Around the same time as the Fedora signing key compromise, Red Hat had one of its systems used for package signing compromised. The attacker was able to sign multiple malicious versions of the OpenSSH software with the Red Hat signing key [46]. Red Hat believes the key itself was not compromised because they use custom hardware requiring physical access to the machine in order to obtain the private key. However, once the malicious software was signed, there was no secure way to revoke the key or the client's trust of the malicious, signed packages.

4.3 Crypto and Implementation Weaknesses

In addition to theft of keys, cryptographic weaknesses also pose a danger to software update systems. There can be a degree of control over some of these threats. For example, implementers of software update systems can choose widely trusted cryptographic algorithms to lessen the likelihood of algorithmic weaknesses. Similarly, the ability for an attacker to derive a private key through brute force methods can be hampered by using large key sizes.

However, there are areas of cryptographic weaknesses that these systems have much less control over. One major problem is that implementation flaws in crypto libraries can render cryptographic algorithms insecure. Such flaws even include the vulnerability of SSL keys to side-channel attacks [9]. Additionally, the discovery of implementation problems resulting in weak key generation has occurred multiple times and has affected thousands of actively used keys [12, 28, 58]. In these situations, it is often the case that the private key can be quickly recovered.

5. ANALYSIS OF CURRENT SYSTEMS

In this section, we look at how software update systems in use today secure the classes of information identified in Section 3.2. There is a large amount of similarity in the security properties of these systems, which we summarize in Table 2. Therefore, we will focus on the general properties and discuss in more detail the notable differences.

Current systems authenticate the content of updates in three primary ways: using SSL, signed files, and with Authenticode. Though Authenticode, Microsoft's form of code signing, uses signed files, we make a distinction because Authenticode has the security problems of PKI. Whether the key that signs files is known directly to the client or authenticated via a PKI, all of these systems use a single key's signature on signed files. From the standpoint of immediate danger due to key compromise, Authenticode and SSL are strictly worse than direct key trust because, as discussed in Section 4, there are more keys that can be compromised as well as other equivalent attacks.

Update availability information is generally either authenticated using SSL or not at all. No update systems provide or encourage methods of ensuring reasonable timeliness of update information using signed files.

The correctness of the repository state seen by clients affects package managers and library managers more than application updaters. Repositories for application updaters often contain only a small number of files (and sometimes a single metadata file), whereas package and library managers typically need to handle hundreds or thousands of packages at a time. Some package managers do provide repository state information through SSL or signed metadata. However, many package managers do not. Also unfortunate is that none of the programming language library managers we looked at [16, 20, 44, 48] authenticate repository state information or any other aspects of their updates. Some library managers, however, do have incomplete proposed schemes for signed packages [49].

Google's update system for Windows software, Omaha, offers an alternative to SSL for providing timely update availability information. This alternative cryptographic protocol is the Client Update Protocol (CUP) [14]. CUP provides authenticity and freshness for software updates with a single request and response over an insecure connection. CUP is not a replacement for SSL as it does not provide privacy, client authentication, or prevention of replay of client requests (it does prevent replay of server responses). The

	Update Content	Update Timeliness	Repository State
yum (Fedora 10)	signed	unprotected	unprotected
yum (Fedora 11)	SSL + signed	SSL	SSL
yum (CentOS)	signed	unprotected	unprotected
yum (Red Hat)	SSL + signed	SSL	SSL + signed
APT (Ubuntu)	signed	unprotected	signed
YaST (OpenSUSE)	SSL + signed	SSL	SSL
YaST (SUSE Ent.)	SSL + signed	SSL	SSL
slackpkg (Slackware)	signed	unprotected	unprotected
Sparkle	SSL and/or signed	SSL or none	SSL and/or signed
Update Engine (Google)	SSL	SSL	SSL
Omaha (Google Update)	SSL + Authenticode	SSL	SSL
Omaha with CUP	CUP + Authenticode	CUP	CUP
Adobe AIR applications	SSL + signed	SSL	SSL + signed
Firefox (Windows)	SSL + Authenticode	SSL	SSL
Firefox (Mac/Linux)	SSL	SSL	SSL
Firefox extensions	SSL or signed	SSL or none	SSL or none

Table 2: Software update systems and the authenticated information they provide.

CUP specification [14] does not address revocation of the public key that clients trust.

6. DESIGN PRINCIPLES

The software update systems discussed in Section 5 share many of the same weaknesses. These weaknesses stem from common design decisions that largely ignore the potential for key compromise. In this section, we describe methods by which software update systems can resist and recover from key compromise. These concepts are not new but their importance and applicability to software update systems has not been generally understood. They will guide the design of our framework in Section 7.

6.1 Responsibility Separation

A software update system can have different roles that are trusted for different responsibilities. These responsibilities may or may not overlap. By carefully dividing these responsibilities, we can achieve a higher level of security by limiting which attacks are enabled when the key for a single role is compromised. A simple example of responsibility separation can be seen in the two-role designs that use SSL in conjunction with a single signing key. In these designs, the signed data does not provide timeliness information; that is solely the responsibility of SSL.

In addition to clients trusting completely unrelated roles for different responsibilities, another way to achieve separation of responsibilities is through delegation. That is, a role can delegate some of its responsibilities to other roles. Delegation can be performed by providing the client with a certificate signed by the delegating role that describes which responsibilities are delegated and to whom.

The division of responsibilities between roles does not by itself improve key compromise survivability. For example, if the same keys are used for all roles, there is no benefit. However, this separation is valuable when different keys are used as well as when combined with other design principles.

6.2 Multi-signature Trust

One of the most obvious ways to prevent software update systems from being at risk due to key compromise is to require the signatures of multiple keys. Multi-signature trust can be achieved in two ways: threshold signatures and responsibilities shared between roles with separate keys.

A single role can have its resilience to key compromise increased by using a (t, n) threshold signature scheme. That is, the signatures of at least t signers are required out of a set of n potential signers. An attacker who compromises $t - 1$ keys cannot successfully attack any clients.

Distinct from threshold signatures, which use multiple keys within the same role, another approach is to require the signatures of multiple roles. The roles involved in multi-role trust share a responsibility between them. Sharing a responsibility is different from delegating a responsibility. When one role delegates a responsibility to another, there is no multi-role trust that results, as either role is trusted for the responsibility without the other role's signature.

These two multiple-signature approaches, threshold signature schemes and responsibilities shared between roles, can be used together. For example, if role A uses threshold (t_A, n_A), role B uses threshold (t_B, n_B), and there are no keys in common between n_A and n_B, then an attacker would need to compromise at least (t_A, n_A) and (t_B, n_B) keys in order to compromise any responsibilities that these roles share.

6.3 Explicit and Implicit Revocation

There are multiple reasons for revoking keys in software update systems. These range from lost keys, suspected or known weak keys, suspected or known compromised keys, and rotation of project members. We divide revocation into two types: implicit revocation and explicit revocation.

Implicit revocation occurs when trust is revoked automatically when some criteria is met. For example, the signatures on a file may be considered to be expired after a specific date or a key may only be trusted to sign information a set number of times before clients stop trusting the key.

Explicit revocation, on the other hand, requires clients to be told that they should stop trusting keys they currently trust. Explicit revocation can be performed by having the trust-delegating role sign a message indicating that trust should be removed. Explicit revocation is always useful to indicate that specific keys should no longer be trusted. It is important to ensure that explicit revocation mechanisms

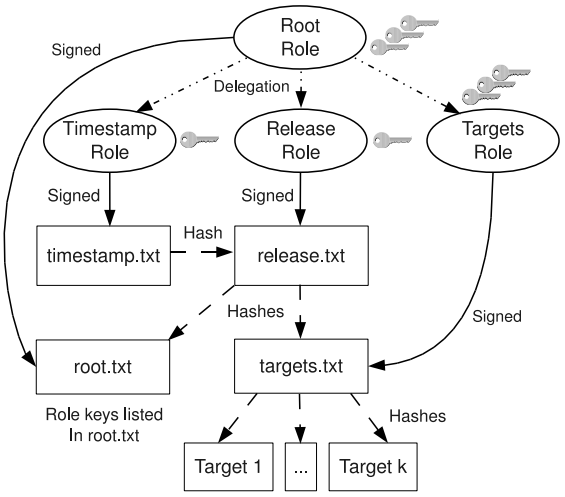


Figure 1: Overview of roles and files in TUF when used with a software update system that does not perform targets delegation. The value of the release role increases when there are delegated targets roles and thus more metadata on the repository.

are not vulnerable to freeze attacks that may prevent clients from being aware of the revocation.

By combining multiple signatures with reliable revocation and replacement of keys, the result is a system that is proactive. Such a design offers resistance to long-term attacks where different keys are compromised over time.

6.4 Minimizing Risk

Key risk is the product of the probability of the key's compromise and the impact of its compromise. When threshold schemes are used, risk can also be considered in terms of role risk, the product of the probability of at least (t, n) keys being compromised and the impact of the role's compromise. The same concept of risk extends to responsibilities shared by multiple roles.

Unfortunately, risk in this context cannot be clearly quantified. The probability of a key's compromise is difficult to determine and it is similarly difficult to quantify the impact of various responsibilities under the control of an attacker. What is clear is that we should decrease overall risk as much as possible. When a responsibility would be very dangerous in the hands of an attacker, the relevant keys should be used and stored in a way that reduces the chance of their compromise. Keys that are stored on systems that are not network-connected are generally much safer than those stored on network-connected systems. Similarly, keys on network-connected systems that are not public-facing are generally much safer than those stored on public-facing servers.

7. DESIGN AND ANALYSIS

The security concepts discussed in Section 6 provide ways to understand and increase key compromise survivability in updaters. This section describes our application of these concepts in The Update Framework (TUF). We distinguish between a full software update system and a software update framework based on whether updates are installed after being retrieved. TUF, being a framework, performs the secure retrieval of updates but leaves installation to the software update system it is integrated with.

For space reasons, some details of TUF's design have been omitted. TUF is the second generation design of Thandy, the updater originally developed for Tor [54]. Detailed descriptions of both are available in the Thandy spec [32] and the TUF spec [56].

7.1 Design Overview

We use the following terminology to describe the design of TUF:

Target files. Target files are the files (updates) that a software update system ultimately wants to download and install.

Metadata files. Metadata files are signed files that describe roles, other metadata files, and target files.

Repositories and mirrors. A repository is a conceptual source of named metadata and target files. Each repository has one or more mirrors. These mirrors are the actual hosts providing the repository's content.

Roles. There is one root role per repository. There are multiple roles whose responsibilities are delegated to them directly or indirectly by the root role. The term *top-level role* refers to the root role and any role delegated by the root role. Each role has a single metadata file that it is trusted to provide.

7.1.1 Repository Contents

`metabase/`	
`root.txt`	
`timestamp.txt`	
`release.txt`	
`targets.txt`	
`targets/foo.txt`	(optional)
`targets/foo/bar.txt`	(optional)
`targetbase/`	
`a.rpm`	(example target)
`x/y.dll`	(example target)

Table 3: Layout of a repository.

The repository contents consist of metadata files and target files. Table 3 shows a sample repository layout. The names of the directories "metabase" and "targetbase" can vary on each mirror. The content and layout of files under those directories, however, is the same on all mirrors of a given repository.

The required metadata files are:

- `root.txt`: Specifies keys of top-level roles.
- `timestamp.txt`: Specifies the latest `release.txt`.
- `release.txt`: Specifies the latest versions of all metadata files other than `timestamp.txt`.
- `targets.txt`: Specifies available target files.

These files are discussed in more detail in Section 7.3.

7.1.2 Client Workflow

Here we outline the workflow of a software update system using TUF to check for and obtain updates. This workflow

shows the steps involved when only the required metadata files are in use. Whenever an unresolvable problem is encountered, the software update system using the framework must decide how to proceed. The updater may want to show information about the problem in a GUI, log the problem, or email an administrator.

Note that all metadata is verified by ensuring the required threshold of trusted signatures, a valid creation time, a future expiration time, and that the file is not older than the last seen version of the same metadata file. With all files except `timestamp.txt`, the client knows and verifies the expected hashes and length of each file.

1. The software update system instructs TUF to check for updates.
2. TUF downloads and verifies `timestamp.txt`.
3. If `timestamp.txt` indicates that `release.txt` has changed, TUF downloads and verifies `release.txt`.
4. TUF determines which metadata files listed in `release.txt` differ from those described in the last `release.txt` that TUF has seen. In the case where no optional metadata files are in use, only `root.txt` and `targets.txt` are listed. If `root.txt` has changed, the update process starts over using the new `root.txt`.
5. TUF provides the software update system with a list of available files according to `targets.txt`.
6. The software update system instructs TUF to download a specific target file.
7. TUF downloads and verifies the file and then makes the file available to the software update system.
8. The software update system installs the update.

7.2 Signed Metadata Format

All metadata files in TUF use canonical JSON [10] format[2] and are enclosed in the signature structure:

```
{ "signed" : X,
  "signatures" : [
     { "keyid" : K,
       "method" : M,
       "sig" : S }
     , ... ]
}
```

where X is the signed object, K is the identifier of the key that created the signature, M is the method used to make the signature, and S is a signature of the canonical encoding of X. An example key format is:

```
{ "keytype" : "rsa",
  "keyval" : { "e" : E,
               "n" : N }
}
```

The signed data of each file contains a creation timestamp and an expiration date. These are the CREATIONTIME and EXPIRES values shown in Table 4.

[2]The specific file format used is not itself important as long as the format is sufficiently expressive and the data can be represented in a canonical form for signing and signature verification purposes.

7.3 Roles and Responsibilities

We use a root role that is responsible for delegating to other roles their responsibility for each type of information. These other top-level roles are the targets role, release role, and timestamp role as shown in Figure 1.

Root role. The root role is the root of trust for the entire repository. The root role signs the `root.txt` metadata file (Table 4). This file indicates which keys are authorized for each of the top-level roles, including for the root role itself. The roles "root", "release", "timestamp", and "targets" must be specified and each has a list of KEYIDs. The signatures of these keys are trusted to sign on behalf of the corresponding role. The signature threshold for each role is the value THRESHOLD. The corresponding public key for each KEYID is specified in the "keys" object.

The keys belonging to the root role are intended to be very well protected and used with the least frequency of any keys in the framework.

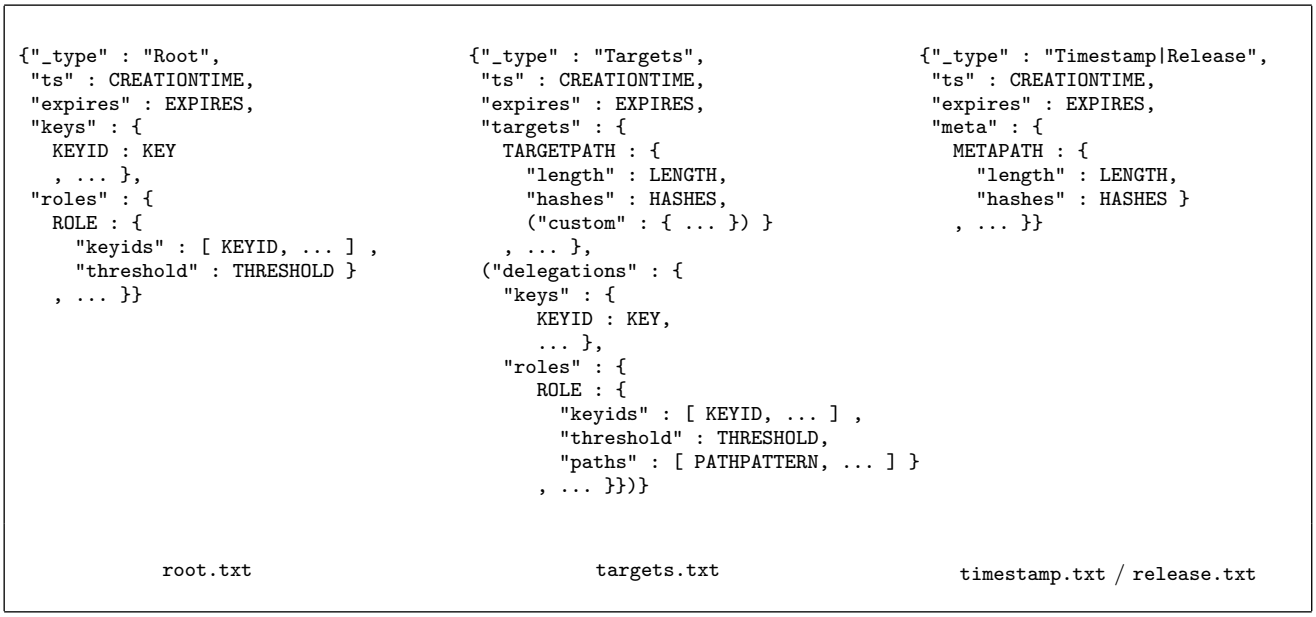
Targets role. Through the `targets.txt` file, the targets role is responsible for indicating which target files are available from the repository. More precisely, the targets role shares the responsibility of providing information about the content of updates. The two roles it shares this responsibility with are the release role and the timestamp role. This sharing is a result of the release and timestamp roles needing to indicate the latest version of `targets.txt` in their metadata files (indirectly in the case of the timestamp role), as shown in Figure 1. Depending on whether the release and timestamp roles are used in an automated fashion, the targets role may have essentially full responsibility for the content of updates. The automated use of roles will be discussed in Section 8.

In the format of `targets.txt` shown in Table 4, each key of the TARGETS object is a TARGETPATH. A TARGETPATH is a path that is relative to a mirror's location of target files. Each element of HASHES is the name of a hash algorithm and the corresponding value is the hex encoded digest of the file's contents. LENGTH is the size, in bytes, of the target file. If defined, the information in "custom" will be made available to the code using the framework; this optional data can be used to indicate additional information such as a file's version number.

The targets role may delegate to other roles the responsibility for providing some or all target files. These delegated target roles are specified in the "delegations" object. The target paths that a delegated role is given responsibility to provide are specified.

Delegated targets roles (optional). If the top-level targets role performs delegation, the resulting delegated roles can then provide their own metadata files. If a delegated role is named *foo*, then *foo*'s metadata file is available on the repository as `targets/foo.txt`. The format of the metadata files provided by delegated targets roles is the same as that of `targets.txt`. Delegated targets roles may also further delegate. As with `targets.txt`, the latest versions of metadata files belonging to delegated targets roles are described in the release role's metadata.

Release role. The release role is responsible for ensuring that clients see a consistent repository state. It provides repository state information by indicating the latest versions of all metadata files on the repository in `release.txt` (Table 4), which it signs. The only metadata file not listed in



```
{"_type" : "Root",
 "ts" : CREATIONTIME,
 "expires" : EXPIRES,
 "keys" : {
   KEYID : KEY
   , ... },
 "roles" : {
   ROLE : {
     "keyids" : [ KEYID, ... ] ,
     "threshold" : THRESHOLD }
   , ... }}
```

root.txt

```
{"_type" : "Targets",
 "ts" : CREATIONTIME,
 "expires" : EXPIRES,
 "targets" : {
   TARGETPATH : {
     "length" : LENGTH,
     "hashes" : HASHES,
     ("custom" : { ... }) }
   , ... },
 ("delegations" : {
   "keys" : {
      KEYID : KEY,
      ... },
   "roles" : {
      ROLE : {
        "keyids" : [ KEYID, ... ] ,
        "threshold" : THRESHOLD,
        "paths" : [ PATHPATTERN, ... ] }
      , ... }})}
```

targets.txt

```
{"_type" : "Timestamp|Release",
 "ts" : CREATIONTIME,
 "expires" : EXPIRES,
 "meta" : {
   METAPATH : {
     "length" : LENGTH,
     "hashes" : HASHES }
   , ... }}
```

timestamp.txt / release.txt

Table 4: Metadata formats. Each is wrapped in the signature structure described in Section 7.2.

`release.txt` is `timestamp.txt`, discussed below, which is always created after `release.txt`.

Timestamp role. The timestamp role is responsible for providing information about the timeliness of available updates. Timeliness information is made available by frequently signing a new `timestamp.txt` file that has a short expiration time. This file indicates the latest version of `release.txt`. The format of `timestamp.txt` is the same as the format of `release.txt`. As the timestamp role needs to frequently sign a file, it lends itself to automated usage with a single key that is potentially kept on a public-facing server. By placing only this one responsibility in a role that has a high likelihood of compromise, we minimize the resulting risk.

7.4 Key Compromise Analysis

We now consider the ability of this design to survive the compromise of one or more roles. The summary of this analysis is shown in Table 5. Based on this analysis, we will discuss recommendations for signature thresholds for these roles in Section 8.

In the following analysis of key compromise, one can often consider the timestamp role and possibly the release role as not being major obstacles for attackers. That is, they do present obstacles to attackers, but some projects may choose to use these roles in an automated fashion and store their keys less securely.

Root role compromise. If less than the required threshold of root keys is compromised, clients are not at any risk. The compromised root keys can be replaced through a new `root.txt` file.

If more than the required threshold of keys belonging to the root role is compromised, an attacker who can respond to client requests can compromise clients. An attacker would sign a new `root.txt` that lists their own keys as the keys belonging to the other roles and provide all of their own signed metadata to clients.

Targets role compromise. If a threshold of targets role keys are compromised, there is no immediate threat to clients. The attacker cannot cause clients to install malicious software unless the timestamp and release roles are also compromised. That is, due to the shared responsibility between these roles for providing update content, all three roles need to be compromised.

If less than a threshold of targets role keys are compromised and the compromise is noticed, these keys can be securely and reliably revoked by having the root role sign a new `root.txt` file that does not list the compromised keys. The keys of the targets role can also be regularly changed using this same method to proactively defend against unknown key compromises.

Delegated targets role compromise. If delegated targets roles are used, they have essentially the same risk properties as the top-level targets role. The differences are that:

- It may be the case that a compromised delegated targets role is only responsible for a subset of the targets available from the repository. Thus, the role's compromise may result in only a subset of clients (e.g. those on a specific operating system) being at risk if the timestamp and release roles are also compromised.
- In addition to revocation by the delegating role, the compromised keys can also be indirectly revoked by any other role in the delegation chain. Indirect revocation occurs when a delegating role has itself been revoked. As a result, all delegations it performed are revoked.

Release role compromise. If a threshold of release role keys is compromised, there is no immediate risk of compromise or even of attacks on the consistency of repository contents. An attacker must additionally compromise the timestamp role in order to perform metadata inconsistency attacks. Revocation of compromised keys is done through the `root.txt` file.

Role Compromise	Malicious Updates	Freeze Attack	Metadata Inconsistency Attack
Timestamp	no	limited by Rel., Targ., and Root	no
Release	no	no	no
Timestamp + Release	no	limited by Targ. and Root	yes
Targets	no	no	no
Timestamp + Targets	no	limited by Rel., Targ.*, and Root	no
Release + Targets	no	no	no
Timestamp + Release + Targets	yes	limited by Root	yes
Root	yes	yes	yes

Table 5: Vulnerabilities when roles are compromised in TUF. The duration of freeze attacks is bounded by the expiration times of metadata signed by uncompromised roles. *Without the release role compromised, the attacker cannot provide their own targets role metadata.

Timestamp role compromise. The compromise of the timestamp role allows freeze attacks on clients. Once the compromise is detected, the duration of freeze attacks is bounded by the earliest expiration time of any of the metadata files. As with the other top-level roles, revocation of timestamp keys is done through the `root.txt` file.

8. EXPERIENCE AND DISCUSSION

In this section, we discuss our preliminary experiences with TUF. We also discuss recommendations and best practices that balance the security benefits of multiple roles and keys with the practical needs of organizations. Specifically, we consider organizations with limited ability to store many keys securely and independently.

8.1 Integration Experience

We have done prototype integrations of TUF with two very different software update systems. The first is the application updater for Seattle [50], an application running on end-user machines that allows the safe execution of untrusted code. The second is with PyPI [45] and easy_install [20], the community repository and library management system for Python.

Seattle's application updater is quite simple. It needs to identify all files for which new versions exist as well as files that did not previously exist, download these files, save them to the application's installation directory, and restart any processes that are affected by the changed files. TUF was integrated such that once all available files were successfully downloaded and verified, the existing application updater is then given access to these files. The rest of the application updater's functionality did not need to be modified.

Python's library management system provides a way for many mutually distrustful library developers to make their libraries available to users. Developers upload their libraries to a single repository, PyPI, and end-users run a library management and update utility, easy_install, to download and install new and updated libraries from PyPI. There is currently no security in the update discovery and installation process.

Our prototype integration [55] with PyPI and easy_install makes heavy use of targets delegation and clearly benefits from the existence of the release role. In this system, PyPI owns the root keys and keeps the keys for all top-level roles. PyPI delegates to each developer a targets role that is only trusted to provide the individual libraries maintained by that developer. The developer can provide any version of these libraries, but is not trusted to provide other libraries. There are over 8000 libraries on PyPI. Our integration uses a separate delegated role for each of these libraries. As a result of the large number of delegated targets roles, the release file is approximately 1.5MB in size due to listing the hashes and lengths of each metadata file. The release file changes every time a developer adds a new version of a library. As a result, it will be frequently downloaded by clients. We implemented gzip compression of the release file, which brought its size down to 460KB. This is similar to the size of the initial metadata that the insecure easy_install downloads on each run, which is 446KB. Another option would be to add support for retrieving deltas (diffs) of large metadata files.

Developers create and sign metadata on their own systems before uploading this metadata and the packaged libraries to PyPI. Developers have the option of using thresholds and delegating further. With respect to PyPI's developer key management, we envision a system where developers use a secure web interface to upload their public keys to PyPI. PyPI may choose to enforce additional security measures to ensure requested key changes are legitimate and not the result of compromised account credentials.

8.2 Recommendations

Number of keys. Given the existence of the root role and three other required top-level roles in our design, many keys may need to be managed. It is important for keys to be stored independently—that is, on separate systems and encrypted with different passwords. If too many keys need to be managed by a small organization, it is likely that many keys would be stored together and thus have the security properties of a single key.

The first consideration is that the timestamp role poses little immediate risk if compromised. It will also often be used in an automated fashion and thus highly vulnerable to exposure if the system it resides on is compromised. Given the low risk of just this role's compromise, the timestamp role can use a single key. The release role is similar to the timestamp role in that the immediate risk that results from compromise is low. Some situations may be different, but in general, it is likely that the release role gains little from threshold signatures and thus can use a single key.

The roles that benefit the most from threshold signatures are the root role and the targets role. The root role is very important because its compromise immediately puts all clients at risk. Additionally, once the root role has been compromised, its keys cannot be reliably revoked and replaced without a trusted communication channel. The targets role should also use threshold signatures for similar but slightly different reasons. Even though the immediate risk due to compromise of just the targets role is much less than that of

the root role, the keys for the targets role will be used more frequently. As a result, the targets role's keys will be more susceptible to compromise. Even for small projects, the burden of threshold signatures for these roles does not need to be high. The threshold can be as little as $(t, n) = (2, 2)$ to gain some level of resilience to key compromise.

Diversity of keys. Due to the threats of weak key generation and algorithmic weaknesses discussed in Section 4, the types of keys as well as the libraries used to generate them should vary within a software update system. Our current implementation of TUF only supports RSA keys and only provides one library with which to generate keys. However, one can generate RSA keys using other crypto libraries and use them in TUF.

Automated signing. The timestamp role will always be used in an automated fashion to frequently resign the timestamp file. Depending on the project, the release file may also be signed in an automated fashion. In order to reduce the likelihood of compromise of keys used for automated signing, the signing should take place on a system that is not public-facing. Once signed, the metadata can then be pushed to a public repository.

Metadata expiration times. All of the metadata in TUF includes expiration times. The appropriate lengths of these expiration times depend on the type of metadata file. For the timestamp file, the expiration time should be short. How short depends on the specific system. For example, if repository mirrors synchronize once a day, the timestamp file will not be able to have an expiration time of less than one to two days, depending on when the synchronizations are done. The release file should have an expiration time not much longer than the period between expected changes to the targets role's metadata. Similarly, the targets role's metadata should use expiration times on the order of the expected time until new target files will be made available. The root role's metadata, on the other hand, should generally have a much longer expiration time than any of the other metadata files. If the root role's metadata expires too soon, then clients who had not updated for an extended period of time will not know whether an attacker is replaying the root metadata. Another reason to use long expiration times on the root metadata is that frequent usage of the root keys exposes them to increased potential for compromise. Thus, it is reasonable for some projects to use a multi-year expiration time with the root metadata.

It is worth noting that systems which are vulnerable to attacks that modify the system time, such as may result from insecure communication with a time server, may be vulnerable to freeze attacks due to inability to detect metadata expiration.

Use of SSL. Our design avoided reliance on SSL in order to ensure the design was not dependent on the security of public-facing servers. However, the use of SSL in conjunction with our design offers further defense against freeze attacks as well as an additional layer of key security when the server and PKI are not compromised. One way to gain the benefits of SSL with minimal overhead is to only transmit the timestamp file over SSL.

9. RELATED WORK

Since the introduction of the first threshold cryptosystem in 1987 [18], many new threshold systems have been developed [51]. Often, these schemes have impractical key setup requirements or assumptions [6]. However, it is trivial to construct a multi-signature scheme by having the multi-signature of a message just be a list of signatures [6]. This scheme is what we use in our design. The advantages of this approach include the simplicity of implementation and the safety from rogue-key attacks, which are generally the result of the key setup process of proposed multi-signature schemes. One drawback of this simple scheme is that the signature size grows linearly with the number of signatures.

Like threshold schemes, proxy signature schemes for delegating signing abilities have a long history [41, 57] and the use of rights delegation in computer systems has of course existed much longer. Our design uses delegation by warrant in which signed certificates indicate the public keys to which specific rights are delegated. This system lends itself to the use of ordinary signature schemes rather than special-purpose proxy signature schemes [29].

The signature scheme we use is forward-secure and has the advantage of being proactive. Forward-secure signature schemes, which are key-evolving, need to be able to change the secret key without having to change the public key [1]. We avoid this complication by using a certificate-based system.

Existing approaches to certificate revocation include the use of semi-trusted mediators (SEMs) [8], certificate revocation lists (CRLs), the Online Certificate Status Protocol (OCSP) [39], certification revocation trees (CRTs) [17], and short-term certificates [40] that allow clients to set their own recency requirements [47]. In our design, we make use of multiple revocation approaches, including short-term certificates, to improve revocation against various types of adversaries.

Our work is not concerned with detecting key compromise but rather only with maintaining a secure state both before and after keys are known or suspected to be compromised. There are methods of detecting key compromise [27], but these generally require signature verification to be done online.

Our previous work identified major security flaws in Linux package managers [11]. That work highlighted the ease with which attackers can become mirrors for popular Linux distributions and in some cases even force specific clients to use the attacker's mirror [37]. Previous work by Bellissimo looked at the limited use of authenticated data by software update systems [7]. That work recommended the development of a standard for secure updates.

We have focused on designing a security framework that is applicable to software update systems on any operating system and for any application. However, some applications and operating systems can also apply the principle of least privilege to installed updates. That is, in addition to decreasing the likelihood of successful attacks on clients by being resilient to key compromise, these systems can also isolate compromises by not giving updates and programs run from updates more privilege than required. For example, web browsers that install unprivileged extensions [5, 38] can keep the rest of the client's system secure even if an attacker causes a malicious extension update. If the installed software is isolated from the software update system sufficiently that the attacker cannot interfere with future updates, a future update could return the system to a secure state. Operating system security mechanisms such as privileged users installing updates for unprivileged users as well

as fine-grained mechanisms such as SELinux can be used for this purpose.

10. CONCLUSION

Software update systems that do not authenticate updates have received increased scrutiny in recent years. Due to this attention, many of these systems have implemented simple authentication mechanisms that cannot survive key compromise. We feel that software update systems must move to using the approaches we advocate in order to be resilient to the many threats to key security. Our open source software update framework, TUF, allows both new and existing systems to benefit from a design that leverages responsibility separation, multi-signature trust, trust revocation, and low-risk roles.

Acknowledgments

We thank those who contributed to Thandy, including Sebastian Hahn and Martin Peck. We are grateful to Geremy Condra, Tadayoshi Kohno, Wenjun Hu, and the anonymous reviewers for their valuable comments.

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0737890 and CNS-0959138, and by Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of BBN Technologies, Corp., the GENI Project Office, or the National Science Foundation.

11. REFERENCES

[1] M. Abdalla and L. Reyzin. A new forward-secure digital signature scheme. *Advances in Cryptology - ASIACRYPT 2000*, pages 116–129, 2000.

[2] Francisco Amato. ISR-evilgrade. `http://www.infobyte.com.ar/down/isr-evilgrade-Readme.txt`.

[3] Vulnerability note VU#944335 Apache web servers fail to handle chunks with a negative size, Jun 2002. `http://www.kb.cert.org/vuls/id/944335`.

[4] APT HOWTO. `http://www.debian.org/doc/manuals/apt-howto/`.

[5] A. Barth, A.P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proc. of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, 2010.

[6] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 390–399, New York, NY, USA, 2006. ACM.

[7] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *1st USENIX Workshop on Hot Topics in Security*, pages 37–43, Vancouver, Canada, Jul 2006.

[8] D. Boneh, X. Ding, G. Tsudik, and C.M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, page 22. USENIX Association, 2001.

[9] Dan Boneh and David Brumley. Remote timing attacks are practical. In *Proc. 12th USENIX Security Symposium*, Washington, DC, Aug 2003.

[10] Canonical JSON - OLPC. `http://wiki.laptop.org/go/Canonical_JSON`.

[11] Justin Cappos, Justin Samuel, Scott Baker, and John Hartman. A look in the mirror: Attacks on package managers. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 565–574, New York, NY, USA, 2008. ACM.

[12] CERT/CC. CERT advisory CA-2000-09 flaw in PGP 5.0 key generation, May 2000. `http://www.cert.org/advisories/CA-2000-09.html`.

[13] Bug 476766 - add China Internet Network Information Center (CNNIC) CA root certificate. `https://bugzilla.mozilla.org/show_bug.cgi?id=476766`.

[14] Open client update protocol. `http://omaha.googlecode.com/svn/wiki/cup.html`.

[15] Microsoft Corporation. Microsoft security bulletin MS01-017, Mar 2001. `http://www.microsoft.com/technet/security/bulletin/MS01-017.mspx`.

[16] CPAN. `http://www.cpan.org/`.

[17] Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. `http://tools.ietf.org/html/rfc5280`.

[18] Y. Desmedt. Society and group oriented cryptography: A new concept. In *Advances in Cryptology - Crypto 1987*, pages 120–127. Springer, 1987.

[19] Vulnerability note VU#800113 multiple DNS implementations vulnerable to cache poisoning. `http://www.kb.cert.org/vuls/id/800113`.

[20] EasyInstall - the PEAK developers' center. `http://peak.telecommunity.com/DevCenter/EasyInstall`.

[21] New signing key. `https://fedoraproject.org/wiki/New_signing_key`.

[22] Firefox update. `http://www.mozilla.com/en-US/firefox/update/`.

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

[24] Omaha (google update). `http://code.google.com/p/omaha/`.

[25] Update Engine and security: How to use Update Engine in a secure manner. `http://code.google.com/p/update-engine/wiki/UpdateEngineAndSecurity`.

[26] Microsoft security bulletin MS08-006 - important vulnerability in internet information services could allow remote code execution (942830), Feb 2008. `http://www.microsoft.com/technet/security/bulletin/ms08-006.mspx`.

[27] M. Just and P.C. van Oorschot. Addressing the problem of undetected signature key compromise. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*. Citeseer, 1999.

[28] Werner Koch. [Announce] GnuPG's ElGamal signing keys compromised, Nov 2003. `http://lists.gnupg.org/pipermail/gnupg-announce/2003q4/000160.html`.

[29] M. Mambo, K. Usuda, and E. Okamoto. Proxy signatures: Delegation of the power to sign messages. *IEICE Transactions on Fundamentals of Electronics,*

Communications and Computer Sciences, 79(9):1338–1354, 1996.

[30] Moxie Marlinspike. Defeating OCSP with the number 3, 2009. http://www.thoughtcrime.org/papers/ocsp-attack.pdf.

[31] Moxie Marlinspike. Null-prefix attacks against SSL certificates, 2009. http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf.

[32] Nick Mathewson. Thandy: Automatic updates for Tor bundles. https://git.torproject.org/checkout/thandy/specs/thandy-spec.txt.

[33] Nick Mathewson. Thandy: Secure update for Tor - Google open source blog. http://google-opensource.blogspot.com/2009/03/thandy-secure-update-for-tor.html.

[34] Extension versioning, update and compatibility: Securing updates. https://developer.mozilla.org/en/Extension_Versioning%2c_Update_and_Compatibility#Securing_Updates.

[35] Microsoft SSL library remote compromise vulnerability (ms04-011, exploit), Apr 2004. http://www.securiteam.com/windowsntfocus/5CP0L0KCKO.html.

[36] Greg Miller. Revving software with Update Engine, Sep 2008. http://googlemac.blogspot.com/2008/09/revving-software-with-update-engine.html.

[37] Mirror manager security risks. http://fedoraproject.org/wiki/Mirror_manager_security_risks.

[38] Mozilla Labs Jetpack | Exploring new ways to extend and personalize the Web. https://jetpack.mozillalabs.com/.

[39] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. RFC2560: X. 509 Internet public key infrastructure online certificate status protocol-OCSP. *Internet RFCs*, 1999.

[40] M. Naor and K. Nissim. Certificate revocation and certificate update. In *in Proceedings of the 7th USENIX Security Symposium*, 1998.

[41] B.C. Neuman et al. Proxy-based authorization and accounting for distributed systems. In *International Conference on Distributed Computing Systems*, volume 13, pages 283–283. Citeseer, 1993.

[42] O. Nordstrom and C. Dovrolis. Beware of BGP attacks. SIGCOMM Comput. *Commun. Rev*, 34(2):1–8, 2004.

[43] Vulnerability note VU#102795 OpenSSL servers contain a buffer overflow during the SSL2 handshake process, Aug 2002. http://www.kb.cert.org/vuls/id/102795.

[44] PEAR - PHP Extension and Application Repository. http://pear.php.net/.

[45] Python package index : Pypi. http://pypi.python.org/pypi.

[46] Critical: openssh security update, Aug 2008. http://rhn.redhat.com/errata/RHSA-2008-0855.html.

[47] R. Rivest. Can we eliminate certificate revocation lists? In *Financial Cryptography*, pages 178–183. Springer, 1998.

[48] RubyGems manuals. http://docs.rubygems.org/.

[49] Signing your gems - RubyGems user guide. http://docs.rubygems.org/read/chapter/21.

[50] Seattle: Open peer-to-peer computing. http://seattle.cs.washington.edu/.

[51] V. Shoup. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000*, pages 207–220. Springer, 2000.

[52] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. Technical Report 684, Indiana University Computer Science Department, April 2010.

[53] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today: Creating a rogue CA certificate, Dec 2008. http://www.win.tue.nl/hashclash/rogue-ca/.

[54] Tor: anonymity online. http://www.torproject.org/.

[55] Securing python package management - tuf: The update framework. http://www.updateframework.com/wiki/SecuringPythonPackageManagement.

[56] /specs/tuf-spec.txt - TUF: The Update Framework. https://www.updateframework.com/browser/specs/tuf-spec.txt.

[57] V. Varadharajan, P. Allen, and S. Black. An analysis of the proxy problem in distributed systems. In *1991 IEEE Computer Society Symposium on Research in Security and Privacy, 1991. Proceedings.*, pages 255–275, 1991.

[58] Florian Weimer. [security] [DSA 1571-1] new openssl packages fix predictable random number generator, May 2008. http://lists.debian.org/debian-security-announce/2008/msg00152.html.

[59] YaST - openSuSE. http://en.opensuse.org/YaST.

[60] Yum: Yellow Dog Updater Modified. http://linux.duke.edu/projects/yum/.