Updatechain: Using Merkle Trees for Software Updates

halpin (Harry Halpin)

May 11th 2016

1 Introduction

The underlying problem of software updates is: How can a user detect if they have been given a malicious software update? For example, in the Apple vs. FBI court-case, the FBI claimed it needed access to a master signing key from Apple to install a update that would let them unlock an iPhone. However, there is a concern that the FBI could then use its access to the master signing key to send updates with targeted backdoors to users [9]. The key problem is that software updates rely almost entirely on a trusted server delivering updates that are validated only by checking a signature. In this paper, we propose a solution called Updatechain that creates an audit log of the binary files in an update using a Merkle tree. We also allow users to verify the chain by relying on multi-signatures from multiple witnesses that maintain their own copy of the Merkle tree in case an updatechain becomes controlled by the adversary. We overview in Section 2 the motivation for more security in software updates. Section 3 details a threat model that can successfully compromise current signature-based approaches and presents a security policy to prevent these attacks. Related approaches like The Update Framework [6] and Binary Transparency [10] are reviewed in Section 4. Section 5 overviews the design of Updatechain and shows it fulfills the security policy. Section 6 presents an API for adding an update, witnessing an update, and verifying updates, including mediations for targetted attacks that rely on Tor and replay attacks that build on the Bitcoin blockchain. Lastly, we detail next steps in Section 7 and conclusions in Section 8.

2 Motivation

Due to the FBI vs. Apple case, there has been increasing interest who controls the keys to sign, and thus authorizes the installation of software updates. A software update is software that modifies existing software on a user's machine through an authorized channel. Although software is continually updated, there has been little analysis of securing the update process itself. The problem with software updates as although the channel is authorized and trusted by the end-user, since an update by nature has a high level of privilege, subverting an update mechanism is an obvious security vulnerability that can be exploited. For example, some are worried about "backdoors" that allow some malicious behavior to be delivered to a user.

In the case of Apple, it is clear that a backdoor was refused. Yet in the other high profile Juniper case, a hack was put into firmware that, likely by changing constants in the Dual EC DRBG generator, allowed keys used by the Juniper VPN to be predicted and so passively gathered VPN traffic to be decrypted.[1] One interesting aspect of the case is that while it was reported the vulnerability was present for over three years, it is unclear at what precise version the vulnerability first appeared. Although a security bug was fixed, there was no audit log to allow the different versions of the Juniper firmware to be audited, so that Juniper users could detect at what time a silent upgrade to their firmware inserted a backdoor into their systems. Thus, it seems some form of cryptographically-verified audit log of all updates for a given application would be useful to discover the exposure to a malicious update.

The FBI vs. Apple may only be the tip of the iceberg, and these requests for backdoors in software delivered via updates may be increasing. For example, the LEAP Encryption Access Project is an open-source project for end-to-end encrypted e-mail aimed at high-risk activists who use the *riseup.net* e-mail provider.[2] When the LEAP Encryption Access Project attempted to incorporate, they were given a request for a "back door key" that they refused based on the reason that "it is well established in cryptographic research that such back doors only undermine the security of the system and lead to eventual exploitation by a nefarious attacker."[3] Even if the owner of a software package refuses to co-operate with a backdoor request, a determined attacker would then attempt to steal the signing keys for software updates or attack the channel the update is delivered over. In order to prevent these kinds of attacks, LEAP is now using The Update Framework (See Section 4.1 for a description) in order to prevent subverted updates.

However, often attacks are not on all users of software, but on one or more targeted users. These users are typically selected based on their IP address, including the use of ranges as IP addresses over geographical areas. For example, it appears that malicious parties committed a targeted attack against Tibetan activists using a malicious update of the popular Android-based KakaoTalk.[4] In fact, now there is a widespread market for signing keys of popular Android updates, where the buyer of the signing key then uses the key for some form of malware.[5]

[1] *http://edition.cnn.com/2015/12/18/politics/juniper-networks-us-government-security-hack/*

[2] *http://leap.se*

[3] Although the full proceedings of this request for a backdoor have not yet been public, the author has viewed the request.

[4] *https://blogs.mcafee.com/mcafee-labs/tibetan-activists-targeted-with-more-android-malware/*

[5] *http://www.androidpolice.com/2011/03/01/the-mother-of-all-android-malware-has-*

3 Problem Definition

The problem with software updates is although the channel is authorized and trusted by the end-user, since an update by nature has a high level of privilege, subverting the update mechanism allows malicious code access to the same privilege level of the application, an exceptional danger in the case of application environments that are not sandboxed. With powerful privilege levels, the malicious update could install a backdoor on the user's computer or otherwise cause considerable harm.

There are a number of principals at work in a software update. A software *update* is software that modifies existing software on a user's machine through an authorized channel. This channel can be anything, including installing updates via portable hard disks or distributed via a peer-to-peer protocol such as Satori.[6] Typically the channel should be an authenticated and encrypted TLS session, although often plaintext HTTP is used to deliver updates. There is an *update server* that hosts the updates. This update server pushes the software update to one or more *clients* (the device of a user) on behalf of the *owner* of the update (which may be the application developer, a package maintainer, a trusted repository manager, or the like).

Although large amounts of software is updated every day and many different systems, the vast majority of update systems simply use the verification of a signature by the owner's *signing key* on the update to secure the update process. The precise owner may vary by the kind of system. Some update models require each application developer sign their own code with a digital signature (Android) while other systems require the app-store itself sign all the updates (Apple). Other systems are mixed (Windows) and some establish a PKI chain of trust from a master signing key to all updates (Linux). Normally, the digital signature may be verified before installation, but some systems allow this verification to be manually overridden (Linux, Windows). In many cases the update server may be a third-party that may or may not have access to the signing key but simply mirrors the signed updates. Some update servers also publish a *checksum* (the hash of the binary) on the update server in order to let the user check the integrity of the update, although verification of the checksum may also not be enforced and currently many checksums use a hash function such as MD5 that is not collision-resistant. As stated earlier, the central problem is: How can a user detect if they have been given a malicious software update? The most obvious answer is to ensure that the update does indeed come from the owner, which can be determined via a signature from the owner's signing key.

Despite this obvious solution, it is exceedingly difficult to determine if the owner themselves is malicious. One of the only recourses is a *witness* (such as a security audit by a trusted party) that verifies that the update indeed fulfills its intended purpose. Another for feasible problem to solve is whether or not a user can determine if other users have received the same update? Users may

arrived-stolen-apps-released-to-the-market-that-root-your-phone-steal-your-data-and-open-backdoor/

[6] *https://play.google.com/store/apps/details?id=com.satori.Satori*

want some level of customization or localization, but in general the user will want to verify that the update they are not being singled out for attack. Lastly, the owner would want to receive the latest update, and not miss a new update. In summary the problem is that the user's client wants to be able to verify that they have received the latest non-malicious update via an honest channel.

3.1 Threat Model

In our threat model, the attacker's goal is to install a subverted update carrying malicious code via a channel onto one or more clients. We assume the owner of an update server is initially honest and that there existed an honest channel to the client such that at some point (the initial installation of the software), the correct public key of the owner was given to the client for signature verification of the updates. We assume an attacker is skilled at the subversion of systems and so can block channels and subvert update servers or any witness of an update server but does not have any special cryptographic powers that allow them to forge signatures or otherwise subvert cryptographic primitives. There are two distinct kinds of attackers.

The most powerful attack is called a *subverted server attack*, the attacker has successfully compromised the signing key of the owner and can send malicious updates to all clients. The updates would be verified as signed by the correct private key of the owner as the attacker controls the signing key. If an attacker did not seize the private keys of the owner, we assume the client could detect the fake signatures and reject the updates. In a variation on this attacked called a *targeted attack*, the attacker may also decide to send malicious updates to one or more targeted clients, where this subset of all clients is chosen via some criteria. In this attack all the updates are correctly signed, but that different clients are receiving different updates signed by the same valid owner's key. If this attack is impossible, the attacker would then try to control the channel via *a man-in-the-middle attack*, which would allow a malicious update to be presented for installation. This man-in-the-middle attack would be equivalent to the subverted server attack if the attacker controlled the signing key, although it does not require the update server itself to be compromised, only the signing key of the owner to be compromised by the attacker. If the signing key was not compromised, the client would reject the updates.

The less powerful attack is a *replay attack*. If an attacker cannot control either the update server or the channel, the attacker can simply block the channel and so prevent new updates from reaching the user or delay the updates in. There are many versions of a replay attack, including a "rollback attack" where the client is tricked into installing an older version of the software (as could be done via altering metadata about the update's date of release) or an "infinite install" attack where an update installation begins but never finishes. The goal of the attacker in these instances is usually to keep the client running older software so they can exploit vulnerabilities that that software.

3.2 Security Goal and Policy

The security goal of our system is to provide the properties that must hold so that a client can verify that the update they received is the same as a non-malicious update from the update server intended to be sent from the owner of the update server. A number of security policies must then hold between the principals:

- To prevent subverted server attacks, one or more witnesses should be able to verify that an update is non-malicious.
- Only the owner should have access to a signing key for the update server.
- The witnesses can attest to their verification of the update by signing the update with their own key and keeping its own copy. Each witness has its own separate key and only the witnesses have access to their witness key.
- To prevent man-in-the-middle attacks, there is at least one honest channel between a witness and the client.
- Each file in an update must be able to have its integrity checked by the client and by witnesses.
- Each update must have a timestamp of its release that the client can verify.
- To prevent replay attacks, the client should be able to discover what the latest version of an update is.
- In case a malicious update is discovered, the client should be able to discover when the update was released.

4 Literature Review

4.1 The Update Framework

One of the primary failures of update systems is lost of key material, leading to the powerful subverted server attack. The Update Framework (TUF) introduces improved privilege separation to the update server that tries to improve over the one master signing key by introducing two new types of keys with explicit trust delegation [6]. Also, to avoid attacks TUF attempts to avoid any external PKI, but instead hosts the entire key structure for a repository in a single *metadata file* that specifies the latest version of the update that may contain multiple files called *target files*. There are five roles: 1) A root role that delegates trusted keys for the client and delegates keys to the rest of the roles. 2) A targets role that signs the metadata for each target file (note that one may have many target roles, up to n for n target files. 3) A snapshot role that signs the metadata of what is the 'latest' update of target files. 4) A timestamp role that signs a hash of each snapshot periodically to confirm the time. 5) A mirrors role that allows one or more *mirrors* to hold copies of each target file and the metadata file, with the total number of mirrors m and number of authoritative mirrors required for installation n specified in the metadata file. Later versions of TUF have developed delegation schemes for the various roles [3].

Each role has a different key, with the root key necessary to keep private as it signs the rest of the keys. Hashes are done via SHA256 and signatures via ECDSA or Ed25519. In order to install a software update, the client periodically downloads the timestamp to see if the snapshot has changed. If the time of the last timestamp update is more frequent than the last update, then it downloads the snapshot file and then iterates through each of the target files, downloading them as well. The client may check using threshold signatures from n of m mirrors. TUF attempts to prevent replay attacks via the use of the snapshot and timestamp file, and prevents key compromises by separating the keys between various roles and then (optionally) requiring threshold signatures between mirrors.

While TUF is definitely an improvement over standard software updates with a single authoritative key, it has no audit log of all updates and only maintains the latest snapshot. Furthermore, TUF does not have any explicit way to let a user determine if they are the victim of a targeted attack. Mirrors help maintain at least one honest channel, but the mirrors are simply copies of the entire update repository, and so the copying of the update repository is left undefined, so the mirrors could copy a subverted update server. Since there is explicit publishing of a timestamp and snapshot, a replay attack is more difficult but not impossible, since an attacker could still block the channel to timestamp and snapshot, especially for a targeted attack.

4.2 Binary Transparency

One of the problems facing TUF is that if there is a malicious update, there is no audit log to discover when the update was released. The first version of BinTrans was started by Huawei to focus primarily on firmware updates due to issues like the aforementioned Juniper attack, but the latest versions attempt to generalize for all possible binaries. Binary Transparency (BinTrans) is a recent and still under development IETF draft that specifies the usage of Merkle trees for audit logs with binary updates, where the leaves of the Merkle tree are hashes of the binaries in an update [10]. It takes inspiration from the more well-known Certificate Transparency IETF standard that uses Merkle trees to provide an audit log for certificates used in TLS [4]. BinTrans currently specifies that CertTrans logs should be used for BinTrans logs. BinTrans specifies a Merkle Tree of the hash (SHA256) of a binary and a signature (either NIST P-256 or RSASSA-PKCS1-V1_5 with at least 2048-bit keys) with a time-stamp. However, while the specification is a straightforward extension of Certificate Transparency, it does not take into consideration the privilege separation of TUF in terms of keys. It is also does not take into account attacks on the channel to the server hosting the BinTrans and so it does not prevent targeted attacks. It does not have a notion of witnesses, although it could depend on the CertTrans gossiping protocol [4]. Similar to TUF, there is still no way to prevent blocking the channel to a BinTrans-hosting server and so prevent replay attacks.

5 Updatechain Design

The Updatechain design combines the concept of a Merkle tree for hashes of files with the verification of an update by multiple witnesses, and adds explicit defenses to prevent targeted attacks and replay attacks. The Merkle tree of the hashes of the binaries in an update is called the *Updatechain* of the update. A single update server may have its own updatechain per server, and each updatechain may have multiple witnesses that try to verify the update and co-sign each verified hash in updatechain. The first innovation of Updatechain is the use of multi-signatures of the witnesses in addition to signature of the update server itself, where all witness signatures must be verified for the installation to continue (as otherwise one of the witnesses or the update server has been compromised), and so detect the subverted server attack. The second innovation is the use of Tor, an anonymizing relay, to provide a trusted channel to the Updatechain and so prevent targeted attacks [8]. The last innovation is the use of the Bitcoin blockchain to store the timestamped and signed Merkle root of the update, allowing a peer-to-peer verification of the Updatechain and detection of replay attacks [5].

The overall design is illustrated in Figure 1, where a malicious update is sent to the client (targeted attack or subverted server) or the channel is blocked via the "blue" arrow and a check to the update server's updatechain is done using the' "purple" arrow. Note that after these two steps are complete, the client has the option of preventing attacks by checking an updatechain via Tor and the timestamp on Bitcoin via the "black" arrows.

5.1 Updatechain and Witnesses

Each update server has an audit log of all their updates. In detail, the update server maintains an *updatechain* for each update U where there are n binary files u in the update ($u \in U$). Similar to BinTrans, the updatechain of each update server is a Merkle tree containing of the hashes of each binary ($h(u)$) using SHA256 for every leaf. The Merkle tree provides an audit log that is easy to check for an update in *O(log n)* time with a corresponding Merkle root for checking the whole tree. Similar to TUF, each update sever has a server's signing key k^r that is derived from an owner's signing key. Although Updatechain like TUF allows separate keys to be used for the roles of each particular target file, version(snapshot), and timestamp, for purposes of a clear exposition we will simply use the server's key as all the keys with roles in TUF are derived from the master key of the owner, although in practice privilege separation should be used between each update and the server, as well as possibly metadata. As in TUF, each update contains metadata, including a timestamp t, a version number v, and the list of all files in the update. When there is a new version of

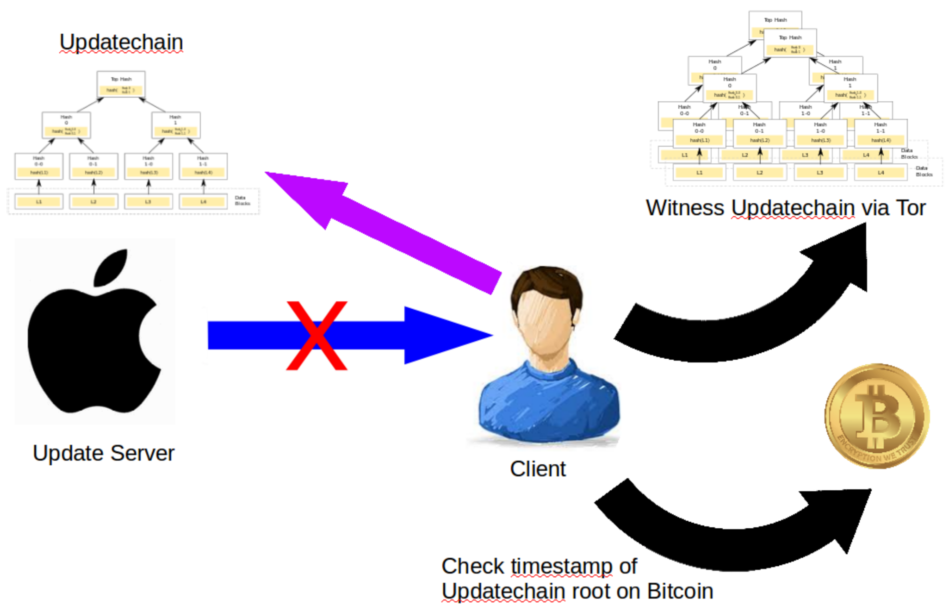

Figure 1: Updatechain Architecture

an update, in order to form a "chain" of updates, we assume the latest version v has each of its files added to the updatechain of the previous version $v-1$. The point of the Merkle tree is to provide an easy way for the client and any other party (such as a witness) to verify the integrity of each file in an update. Each leaf in the Merkle tree is signed by k^r. We assume there will be key rotation of k^r and its public key finds its way to the client, and we'll assume a log of all previous keys are kept.

Witnesses exist to prevent subverted server attacks. Each updatechain m may have one or more (l) witnesses $w_1^m...w_l^m$. The group of witnesses is a pre-authenticated group whose task it is verify the updatechain of a server and create a copy in case the original updatechain gets subverted. For example, a trusted group such as EFF or security specialists such as iSec partners could serve as a witness for Google's binaries for Android. The witnesses are a social group that has established a trust relationship with the owner of the update server before the update. The role of the witnesses is to detect a subverted server attack. The canonical behavior of a witness is to take the source code of an update and audit it for malicious behavior, such as a backdoor, and we assume they can replicate the binary on the server and other witnesses via deterministically building from source to binary. By having a group of witnesses, we assume that an attacker will not be able to compromise all the witnesses but only a subset of them and we increase the chances of an audit of the code being successful. As the clients are assumed to be unauthenticated, the witnesses may not be the client itself as an attacker could then 'spam' the updatechain by creating fake witnesses. In addition there may be up to l witnesses with keys $k^w_1 ... k^w_l$ that they use to co-sign a leaf of m if they find no malicious behavior when replicating the build and so the hashes of their binaries match. We also assume a client can verify signatures by having a list of witnesses with each public signing key pk^w and update server key pk^r. As suggested by Syta et. al in terms of witnesses, we would also recommend Schnorr signatures are used as opposed to traditional multi-signatures due to their scaling properties and have well-understood security proofs that do not require features like randomnesses that can be subverted [7].

5.2 Keeping an Honest Channel via Tor

We can imagine the updatechain of a subverted update server or subverted witness providing false update information via targeting the client. Typically, a target client of a targeted attack is identified by their IP address. In geographically bounded attacks, the attacked targets are also identified by a range of IP addresses that stand-in for their geographical area, as an IP address can typically be resolved to at least the geographical area of the ISP (Internet Service Provider) that assigns the IP address. Tor allows the IP address of a Tor client that is sent over the Tor network to be anonymized, and replaced with the IP address of an exit from the Tor network [8]. In order to prevent targeted attacks on a particular client via their IP address when checking the updatechain, the client maintains the honest channel with at least one witness via the use of the Tor anonymizing proxy. At first glance, this would allow the same updatechain to be retrieved from many IP addresses, as theoretically Tor could be considered an approximation for a random IP address. However, in practice Tor users are mostly in the United States and most Tor exit nodes are in Europe.[7] So it is better to have an updatechain accessed via a Tor Hidden Service, which is an Internet service (such as a website) that can only be accessed via the Tor network. Since a hidden service can only be accessed by the Tor network, its own IP address is unknown and it cannot distinguish between the IP addresses of those clients accessing the repository. Also, if the channel to the update server is itself blocked or spoofed via a man-in-the-middle attack, the use of the witness on the Tor hidden service should prevent the attack. This use of a witness on the Tor hidden network should detect targeted attacks in particular, unless the entire Tor network entry and exit nodes were being blocked. However, as Tor is a dynamic network and as there is a considerable "arm's race" between Tor developers and various censors wanting to block Tor, we consider this unlikely and as a client would simply come equipped with the latest Tor proxy access to take advantage of any circumvention software developed by Tor. It does not usually make sense to download all the updates from the update server itself, as Tor incurs considerable latency delays.

[7] *https://torflow.uncharted.software/*

5.3 Finding the Latest Version with Bitcoin

Yet if there is a replay attack that blocks the channel to the update server and all the witnesses, we would still want some way to determine if there is an latest version even if we cannot prevent the attacker from blocking the channel. Updatechain improves on TUF by allowing the timestamp of the update to be verified via the use of the Bitcoin blockchain at low (although not zero) cost [5]. Since the Bitcoin blockchain allows arbitrary metadata to be added via the *OP_RETURN* code,[8] any updatechain server (as well as witnesses and even clients) can simply add to the Bitcoin blockchain the signed SHA256 hash of its Merkle root with a timestamp. So a replay attack cannot be successful unless the attacker is able to successfully fork the Bitcoin blockchain, which is very difficult. In essence, the hashing power of the Bitcoin blockchain provides an additional assurance of the software update's timestamp and the integrity of the updates. This opens the question as to why the entire Bitcoin blockchain is not used to store the entire updatechain or each of the binaries of an update. While this would be theoretically possible, it would involve adding many new transactions to the Bitcoin blockchain, and the cost would grow, while only storing the Merkle root once per update allows a very low cost to be used to store once for the entire update, achieving the same goal. The use of a sidechain may not work, as a determined attacker could much more easily fork the sidechain as it would require less hashing power. The same argument would apply to creating a new alternative blockchain for private updates. Rather than re-invent the wheel, it make the most sense to simply take advantage of the massive hashing power already in the Bitcoin blockchain for providing additional assurance against replay attacks.

6 The Updatechain API

We consider Updatechain to have a update server (repository) r that contains the software update $U = u_1...u_n$ where $u = \{0,1\}^*$ with an updatechain m. Each repository has a secret signing repository key sk^r. There is also a Bitcoin blockchain B that can add new data x in $B_t = B_{t-1}(x)$ as an append-only log, and also return $x = B(x)$ if x is on Bitcoin blockchain. We are assuming the use of Schnorr signatures or another aggregate signature scheme.

6.1 Adding an Update to the Updatechain

m=Add(U,m): When a new update is to be added to m for version v at timestamp t, construct the binary hash tree m and add the Merkle root m_{root} to B.

1. For each $u_x \in U$, apply SHA256 to obtain $h(u_x)$.
2. Sign each leaf x with owner's signing private key $\sigma_{sk^r}(h(u_x))$.

[8] *https://github.com/bitcoin/bitcoin/pull/2738*

3. If n is odd, then an update $u_{n+1} = u_n$ is added to keep the Updatechain balanced.

4. For each leaf $h(u_x)$ and $h(u_{x+1})$, parent nodes $h(w)$ are constructed $h(w) = h(h(u_x) \parallel h(u_{x+1}))$.

5. Repeat previous step until Merkle tree m is built with level z.

6. If there is no previous existing version $v - 1$, then $m_{root} = h(h(z_1) \parallel ...h(z_2))$.

7. If there is a previous version $(v-1)$, then recalculate the $m_{root} = h(m^{v-1}_{root} \parallel m^{v}_{root})$

8. Create signed Merkle root metadata $m_{metadata} = \sigma_{sk^r}(m_{root} \parallel' 0x0' + v \parallel' 0x0' + t))$

9. Add metadata to blockchain $B(m_{metadata})$

6.2 Witness an Updatechain

m=Witness(W,U,m): One or more (l) witnesses $W = w_1...w_l$ with secret signing keys $sk^w_1...sk^w_l$ can verify an updatechain by co-signing based. The leaf x of Merkle tree m is indexed to the beginning of U and denoted m_x. Error messages that show an attack have been discovered are given by *E('name of attack')* as defined in the threat model.

1. For each $w \in W$
   (a) For each file $u_x \in U$, audit u_x.
   (b) If no security flaw is found in u_x and $h(u_x) \neq m_x$, then $\sigma_{sk^w}(m_x)$ else *E(subverted server)*

2. $w = m$

6.3 Verifying an Update

{True,False}=VerifyUpdate(U,m,W): There is a client c (with public signing keys $sk^w_1...sk^w_l$ of the witnesses and public signing key pk^r of the update server) that is trying to determine if their local copy of the update U has the same integrity as m. The leaf x of m is indexed to the beginning of U and denoted m_x. We assume there is a single aggregate signature $(\sigma_W = \sigma_{sk^1}...\sigma_{sk^l})$ of all the witnesses. The usage of Tor for a client and a witness available at a hidden service w^{tor} is given by $w^{tor} = Tor(c, w)$. For verification an error message E should cause the process to abort.

1. $Verify(pk^r, m_{metadata}, \sigma_{sk^r}(m_{metadata}))$ else *E(man-in-the-middle attack)*

2. If $m_{root} = Tor(w_{root})$ else *E(targeted attack)*

3. $B(m_{metadata}) = m_{metadata}$ else *E(replay attack)*
4. For each $u_x \in U$
   (a) If $Verify(pk^r, h(u_x), \sigma_{sk^r})$ and $Verify(pk^r, h(u_x), \sigma_w)$ else *E(subverted server)*

7 Next steps

A "proof-of-concept" of updatechain is still under implementation based on the work of RS-Coin[2].[9] More importantly there is also more work to be done before Updatechain is ready for usage. First, there needs to be performance testing on how the number of witnesses, the use of Tor, and the use of the Bitcoin blockchain impact the practical efficiency of checking updates. In terms of the underlying assumptions, the method has only been detailed insofar as it would apply to the case of a single software update based on a group of binaries that can all be deterministically built and checked by witnesses. In reality, software updates are often much more complex and involve entire interlinked groups of updates from multiple update servers, where the updates may or may not be compiled from source code that can be verified by witnesses. Furthermore, many code-bases also are currently interlinked with dynamic code delivery for the Web: For example, many applications often require registration via a Web form. Thus, in the future the Updatechain approach needs to be scalable, apply to the installation of software from source code rather than binaries, to take into account software dependencies, and work with Javascript from the Web.

7.1 Scalability

Updatechain needs to be scalable to entire distributions of updates, not just single updates. For example, as of 2016 there are possible 69,559 packages in Ubuntu,[10] with a total of *80,000* files on a typical default installation of Ubuntu. Given the number of updates (Ubuntu averaging 3 updates a year of their packages) we need to make sure our approach scales. Note that also there are critical security updates in between version numbers. Over 2015, Ubuntu 15.04 received approximately 100 updates.[11] With new versions being released approximately once a year, we need to be able to scale the approach from a single software package containing ≤ 100 files to entire operating systems approaching $\leq$ 100,000 and trees of version updates that could eventually go over a $\leq$ a million files. One next step is simply to test scalability over much larger collections of files. It is possible the approach may have to be simplified for the auditing of entire distributions.

[9] *https://github.com/hhalpin/updatechain*
[10] *http://packages.ubuntu.com/yakkety/allpackages?format=txt.gz*
[11] *http://www.ubuntu.com/usn/vivid/*

7.2 Source Code

One of the primary issues with our approach is it defines a software update as a binary blob and assumes there to be a deterministic build from the source code to the binary for both the witness and client. Using an updatechain of the hash of the binary only works as long as there is no non-determinism from the compiler (such as many techniques used in compiler optimization) or other non-malicious causes for the binaries not to match (such as localization). The problem of deterministic builds has been actively solved for a fairly large part of the Debian environment by the reproducible build effort.[12] An alternative approach would be to convert the source code of each file into binary, using a technique such as Base64, and then store the hash of the source code in Updatechain. While it is possible an attacker could attack the compiler, it is unlikely.

7.3 Dependencies

Our approach also assumes that a single collection of binary files contains the application. Indeed, this is not often the case, as usually there are underlying dependencies. The use of the secure "Off-the-Record" messaging system given by *libotr* relies on the XMPP library *libpurple*, which has been shown to have security vulnerabilities.[13] In order to give security guarantees, one needs to do operations that check multiple binaries against relevant parameters such as their version number, including not just the downloaded application code but already pre-existing libraries. This kind of simplified interpreter is familiar to blockchains in terms of Bitcoin's Script language, and already new languages specialized in providing guarantees for updates over the Bitcoin blockchain have started to be under development, such as Dex.[14] Since we want Updatechain to be used for general purpose updates, it seems providing a separate interpreter may be less useful than allowing the generic Updatechain API to be integrated into existing installation management scripting languages such as *make* and *configure* for C.

7.4 Web

In reality, many applications are spread throughout a mixture of native code, compiled to binary on a client, and Web-based code that is dynamically ran from a server. However, Javascript code is both not delivered as a binary but delivered as Javascript via a browser and often delivered in an obfuscated fashion, presenting problems for Updatechain. The W3C has recently developed a new standard called Subresource Integrity that allows both a hyperlink tag or a script tag to contain a hash of some Javascript code, and to only execute the

[12] *https://wiki.debian.org/ReproducibleBuilds*

[13] *http://motherboard.vice.com/read/desktops-urgently-need-a-more-secure-chat-program-adium-pidgin*

[14] Personal communication from Peter Todd.

linked Javascript if the hash matches [1]. As they are listed as authoritative for particular Javascript code, the hashes from Subresource Integrity could be added to Updatechain, although they would not be signed by default, but could be explicitly signed by the maintainer of the server or the user who executes the code. An additional future feature is that Updatechain's API could be presented as REST API that could be access by a browser via HTTPS, allowing the installation of Javascript to checked in the browser as part of a Web Application.

8 Conclusions

Updatechain shows that software updates can be done in a Merkle-tree method that is both easily auditable and multi-signatures allow the use of witnesses to verify the veracity of updates, thus giving a user reasonable assurance that they have received the same update as other users, protecting against a number of attacks such as man-in-the-middle attack on the connection between the user and an updatechain and replay attacks that withold an update. In order to detect attacks on targeted users, a number of defenses have been implemented the use of the Bitcoin blockchain for periodic storage of "timestamps" on updates as well as the use of Tor to provide access to an Updatechain that cannot be censored via IP address level control. Although the current approach will be integrated into The Update Framework that is used by "mission critical" software such as LEAP that is likely to be targeted for attack, future work into scalability and integration into deterministic builds should allow Updatechain to be used on entire operating systems.

References

[1] Frederik Braun, Devdatta Akhawe, Joel Weinberger, and Mike West. Subresource integrity, 2016. https://www.w3.org/TR/SRI.

[2] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

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

[4] Ben Laurie, Adam Langley, and Emilia Kasper. Rfc 6962: Certificate transparency, 2013. https://tools.ietf.org/html/rfc6962.

[5] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

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

[7] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Linus Gasser, Nicolas Gailly, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *37th IEEE Symposium on Security and Privacy*, 2016.

[8] Paul Syverson, R Dingledine, and N Mathewson. Tor: the second-generation onion router. In *Usenix Security*, 2004.

[9] Agnidipto Tarafder and Arindrajit Basu. A small battle for fbi, a gigantic war for privacy rights. *Economic & Political Weekly*, 51(17):13, 2016.

[10] Dacheng Zhang, Daniel Kahn Gillmor, Ana Hedanping, and Behcet Sarikaya. Rfc 6962: Certificate transparency, 2016. https://datatracker.ietf.org/doc/draft-zhang-trans-ct-binary-codes/.