CHAINIAC: Proactive software update transparency via collectively signed skipchains and verified builds

OCTOBER 12, 2017 ~ ADRIAN COLYER

CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds Nikitin et al., *USENIX Security '17*

So hopefully you've put in place some kind of software supply chain management process that will pick up the availability of new package versions, particularly of course those with fixes for discovered vulnerabilities, and ensure those updates are propagated through your environment in a timely manner. But have you ever thought about this: with the advent of continual[1] delivery pipelines an attacker has a wonderful opportunity to inject a vulnerability anywhere upstream in your process (from the source code control system onwards) and have that picked up and handily deployed into your production systems for them. This is not just a theoretical risk. So you also need to think about *securing your software supply chain*.

> Software-update mechanisms are critical to the security of modern systems, but their typically centralized design presents a lucrative and frequently attacked target…

CHAINIAC addresses the challenge of securing package update managers – and it turns out to be quite a complex issue when you start digging into it. It's an important one though: what's the one command you probably always run and implicitly always trust: `apt-get upgrade` or its equivalent! Starting from a design typical of today's software update systems, the authors walk us through a seven-stage evolution to arrive at a more trustworthy design. Along the way we're going to learn about reproducible builds, collective signing and cothorities, and a new kind of blockchain-based structure called a *skipchain*. The end destination looks like this:

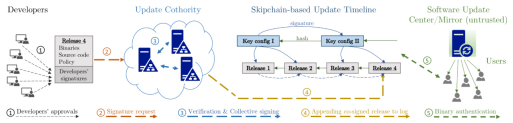


Figure 1: Architectural overview of CHAINIAC

CHAINIAC provides the following properties:

1. No single point of failure – the security guarantees still hold in the event that any single component (software or human) is compromised.
2. Source-to-binary affirmation (what you're running matches source code you can inspect)
3. Efficient release search and *verifiability*.
4. Linear immutable public release history.
5. Support for evolution of signing keys
6. Timely updates – clients can verify that the software really does correspond to the latest one available.

The Starting Point

We begin with a system in which a single key pair is used to sign and verify software releases. The private key is most likely shared among a group of developers, and the public key is installed on client devices. To distribute a new release, one of the developers builds the source code, signs the resulting binary, and pushes it to a trusted software update centre. Users receive authenticated releases with minimal overhead.

> This design, though common, is rife with precarious assumptions. Expecting the signing key to be uncompromisable is unrealistic, especially if shared among multiple parties, as attackers need to subvert only a single developer's machine to retrieve the secret key or to coerce only one of the key owners. For similar reasons, it is utopian to assume that the software update center is trustworthy…

Improvement #1: Decentralised release approval

Instead of using a single shared key to sign updates, each individual software developer signs using their own *individual keys*. Developer's public keys are collected in a policy file, together with a threshold value specifying the minimal number of valid signatures required to make a release valid.

> We assume that this policy file, as a trust anchor, is obtained securely by users at the initial acquisition of the software, e.g., it can reside on a project's website as often is the case with a single signing key in the current software model.

To make a release, developers check the *source code*, and if they approve, sign a hash of it with their individual keys. The source code plus signature list is pushed to the update center. Why source code instead of binaries? It's much easier to verify human-readable code (though still impractical for projects of any size I would argue!), and it is otherwise very hard to verify the mapping between a given set of source code, and a particular binary. We'll restore the convenience of binary updates in the following steps…

Improvement #2: Pre-built binaries you can trust

Asking users to build binaries for themselves is a step backwards in usability. So the second improvement on our journey allows *developers* to build the binaries rather than end-users. After validating the source code, each developer compiles it using *reproducible build techniques*. If the result of a developer's build matches the announced binary, she or he signs the software release. A release now constitutes the source code, binary, and signatures packet.

> Reproducible builds are [sic] software development techniques that enable users to compile deterministically a given source code into one same binary, independent of factors such as system time or build machines. An ongoing collaboration of projects is dedicated to improving these techniques, e.g., Debian claims that 90% of its packages in the testing suite are reproducible…

You can find out more about at reproducible-builds.org.

Improvement #3: Introducing cothority

As things now stand, every developer needs to perform a reproducible build, and client devices need to verify many developer signatures. It would be more convenient if a trusted third-party could take the developers' commitments, run the reproducible build, and produce a result easily verifiable by clients. But 'trusted third-party' is a red flag from a security perspective. So to maintain decentralisation, CHAINIAC implements the intermediary as a *collective authority*, aka., a *cothority*.

> CoSi is a protocol for large-scale collective signing. Aggregation techniques and communication trees enable CoSi to efficiently produce compact Schnorr multi-signatures and to scale to thousands of participants. A complete group of signers, or witnesses, is called a collective authority, or cothority.

Developers send the release data and their signatures to the cothority, which collectively validates and signs the release. Clients can download and validate the release source code and/or binary by verifying only a *single collective signature* and Merkle inclusion proofs for the components of interest.

These *verified builds* give clients the guarantee of source-to-binary correspondence without the resource-consuming building work (and also without needing to install the build toolchain on machines where you simply want to verify and apply updates).

Improvement #4: Anti-equivocation

If a threshold number of developers *could* be coerced into creating a secret backdoored release used for targeted attacks, the protocol as described so far would still be vulnerable. We're talking about very determined or maybe even nation-state attackers at this point.

> In our next step towards CHAINIAC, we tackle the problem of such stealthy developer-equivocation, as well as the threat of an (untrusted) software-update center that accidentally or intentionally forgets parts of the software release history.

Step 4 adds *cothority-controlled hashchains* that create a *public history* of the releases for each software project. This blockchain includes a new block for each public release, and a (signed) new release include the Merkle root of the software's previous version. This makes it impossible for a group to sign a compromised release and keep it off the public record.

> This approach prevents attackers from secretly creating malicious updates targeted at specific users without being detected. It also prevents software update centers from

> "forgetting" old software releases, as everything is stored in a decentralized hash chain.

The collective signature for the new block corresponding to the release is created using the BFT-CoSi protocol introduced in ByzCoin. It implements PBFT using collective signing with two CoSi rounds for PBFT's prepare and commit phases.

Improvement #5: Evolving keys

Key compromise is only a matter of time, so we're going to need the ability to rotate keys – ideally with different schedules for different witnesses. CHAINIAC adds another decentralised mechanism for trust delegation that enable evolution of keys.

> As a result, developers and cothorities can change, when necessary, their signing keys and create a moving target for an attacker, and the cothority becomes more robust to churn.

The trust delegation mechanism is implemented by a new derivative of a blockchain structure the authors call a *skipchain*. Each cothority configuration becomes a block in the skipchain, and when a new cothority configuration needs to be introduced the current cothority witnesses run BFT on it. (It's somewhat analogous to changing epochs in consensus protocols). To rotate developer keys, the project policy file is the root of trust. This is included in the Merkle tree of the release, and hence also protected by the hash chain.

A skipchain combines ideas from blockchains and skiplists, enabling clients to securely traverse the timeline in both *forward* and backward directions, and to efficiently traverse long distances in the chain using multi-hop links. Multi-hop link lengths can be determined randomly or deterministically. In both cases, skipchains enable logarithmic-cost timeline traversal.

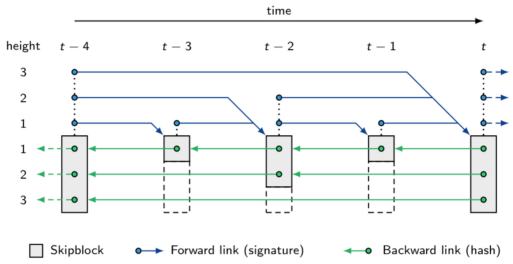


Figure 2: A deterministic skipchain $\mathcal{S}_2^3$

Now, how exactly do you put *forward* links (i.e., links to blocks that haven't been created yet), in a *immutable* data structure?

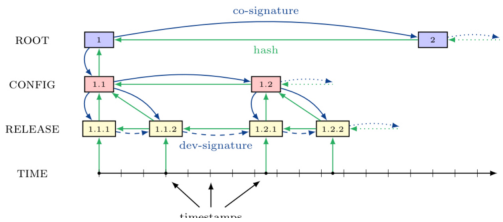
> Forward links are added retroactively to blocks in the log, as future blocks do not yet exist at the time of block creation. Furthermore, the forward links cannot be cryptographic hashes, as this would result in a circular dependency between the forward link of the current and the backward link of the next block. For these reasons, forward links are created as digital (multi-) signatures.

For more detail, see section 4.1 in the paper. The authors note that skipchains may have applications in several other domains requiring efficient timeline tracking.

Improvement #6: Ensuring timeliness

> In addition to verifying and authenticating updates, a software-update system must ensure update timeliness, so that a client cannot unknowingly become a victim of freeze or replay attacks. To retain decentralization in CHAINIAC, we rely on the update authority to provide a timestamp service.

How this is done is kind of neat, with a multi-layer skipchain architecture in which the skipchains are interconnected via upward and downward links:



timestamps

Figure 3: Trust delegation in CHAINIAC

The Root chain is CHAINIACs root of trust – the most security critical signing keys, kept offline. The Config chain holds the online keys of the update cothority, and is CHAINIACs control plane. The Release chain manages the release log as described so far, with the addition of upward links to the root and config chains.

The *time* chain provides a timestamp service that informs clients of the latest version of a package within a coarse-grained time interval (e.g., one hour).

> Every TIME block contains a wall-clock timestamp and a hash of the latest release.

Improvement #7: Multi-package projects

The final enhancement makes support of multi-package projects more efficient, via an aggregate layer that pulls together all of the packages in a project into a single Merkle tree.

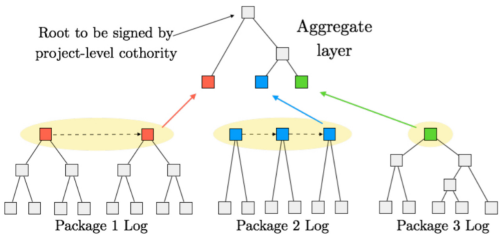


Figure 4: Constructing an aggregate layer in CHAINIAC

Implementation

> We implemented CHAINIAC in Go and made it publicly available, along with instructions on how to reproduce the evaluation experiment. We built on existing open-source code implementing CoSi and BFT-CoSi. The new code implementing the CHAINIAC prototype was about 1.8kLOC, whereas skipchains, network communication, and BFT-CoSi were 1.2k, 1.5k, and 1.8k lines of code respectively. Although the implementation is not yet production quality, it is practical and usable for experimentation purposes.

---

1. That one's for you, @tastapod! ↩

POSTED IN UNCATEGORIZED

SECURITY

< PREVIOUS
TrustBase: an architecture to repair and strengthen certificate-based authentication

NEXT >
BadNets: Identifying vulnerabilities in the machine learning model supply chain

1 comment sort by **relevance**

Sign up Sign in

Start a conversation ...

In-toto: providing farm-to-table guarantees for bits and bytes – the morning paper (guest)

5 years ago

[…] a level of protection and assurance that npm users can only dream of! It's similar in spirit to CHAINIAC that we looked at a couple of years […]

Share Vote Reply

Terms of use - Privacy - Report a bug

powered by GraphComment