Fighting targeted backdoors with secure multiparty code reviews and a SSOT

Frank Braun

2018-10-06

Contents

Reflections on trusting trust

> "To what extend should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people *who wrote the software*."

(Ken Thompson, Turing Award Lecture[1], 1984)

[1] https://doi.org/10.1145/358198.358210

This famous statement leads to the following questions:

- How can we trust the people who wrote the software?
- How can we make sure we actually run the code they wrote?

The problem of targeted backdoors

The Fog of Cryptowar[2] warns "that the picture painted in [the] media as well as by experts and pro-crypto activists, may be misleading and creates the potential to engage a straw man put up by the executive branches in various countries. This response risks that pro-crypto forces miss the big picture of regulation in the communications sphere."

Crypto regulation is **not** black-and-white, the core of the debate is not actually about outlawing crypto.

But often pro-crypto groups focus on such black-and-white scenarios with bad counterarguments like "banning cryptography is like banning math" which misses the point.

Increasing regulations in the cryptocurrency space are a good example for the **practical** implications of regulations: Although the crypto itself isn't outlawed, more and more KYC and AML regulations have huge effects on the practical usability of cryptocurrencies, **especially** for use cases where at least one of the transaction participants needs financial privacy. What's the use of a cryptocurrency if you cannot easily buy or sell it?

Todays widespread use of cryptography to encrypt personal communication and data leads to the situation that the confiscation of computers and smartphones no longer guarantees access to evidence that would stand up in court:

> "It is in light of cryptography providing effective limits to court orders and warrants, and the history of previous regulation to make law enforcement effective, that now regulation on cryptography is demanded.
>
> The main question remaining is if and how cryptography can be regulated without causing too much collateral damage to the societal uses of cryptography.
>
> [...]
>
> The goal is not, and cannot be, to snatch strong cryptography from the hands of people. Instead, the current debate is about making the secrets that cryptography protects accessible to law enforcement. This is no minute point since it deeply shapes the approach that regulators take, and it is therefor the point with which pro-crypto activists must engage. Failing to see that the goal is access to the

[2] http://shadowlife.cc/files/hcpp17-smuggler.html

> plaintext confines arguments into a space that is neither relevant nor commonly understandable for public opinion. While it is certainly difficult to appreciate this difference from the perspective of cryptography it is nevertheless substantial—because it allows for very different technical implementations and legislative action."

(from The Fog of Cryptowar[3], section "Motives for regulating cryptography")

The article then goes on to argue that following five regulatory approaches are both realistic and likely:

1. Defense of metadata access.
2. Nudge vendors to deliver software with less secure default settings.
3. Lawful hacking.
4. Use of update mechanisms to deliver police Trojans.
5. Mandate plaintext access.

In this paper we focus on *technical mitigations* to issue 4, the use of *targeted updates* to introduce backdoors into specific devices in order to surveil the user (so-called *targeted backdoors*).

The relevant suggestions in this regard were:

1. **Secure software distribution**: Automatically verify with some form of single source of truth (SSOT) that a program delivered to a devices is the same as the installation on all other devices.
2. **Secure software development**: Better review and auditing processes for security critical code. Most importantly, enforce that code changes *always* require the cryptographic signatures of *at least two* developers.
3. **Verifiable build processes**: Review and audits are of little use, if the build process (the compilation) cannot be verified. This means that verifiable, deterministic builds should become commonplace.
4. **More decentralized platform vendors**: Today most users depend on very few platform providers that control both the operating system and the application delivery channels, which allows regulators to capture wide sections of the market by focussing all attention on a handful of corporations in a few jurisdictions.

The technical mitigations to targeted updates discussed in this article concern suggestions 1 to 3. Suggestion 4 is more non-technical in nature and is discussed in the conclusion at the end.

[3] http://shadowlife.cc/files/hcpp17-smuggler.html

Recent developments regarding targeted backdoors

Unfortunately, *targeted updates* is exactly the approach the Australian government, a member of the Five Eyes, is taking in their proposed Assistance and Access Bill 2018:

> "[Technical Capability] Notices may still require a provider to enable access to a particular service, particular device or particular item of software, which would not systemically weaken these products across the market. For example, if an agency were undertaking an investigation into an act of terrorism and a **provider was capable** of **removing encryption from the device of a terrorism suspect without weakening other devices in the market** then the provider could be compelled under a technical assistance notice to provide help to the agency by removing the electronic protection.
>
> The mere fact that a capability to selectively assist agencies with access to a target device exists will not necessarily mean that a systemic weakness has been built. The nature and scope of any weaknesses and vulnerabilities will turn on the circumstances in question and the degree to which malicious actors are able to exploit the changes required.
>
> Likewise, a notice may require a provider to **facilitate access to information prior to or after an encryption method is employed**, as this does not weaken the encryption itself. A requirement to disclose an existing vulnerability is also not prohibited."

(from the Assistance and Access Bill 2018, Explanatory Document[4], August 2018, page 47, highlighting added)

If one looks behind the general formulations used in the explanatory document, it becomes clear that *targeted updates* are the concrete technical methods available today ["provider was capable"] which allows service providers to give governments these *targeted backdoors*.

These targeted updates would either work on the application level or the operating system level:

- A targeted update to a secure messaging application could introduce a backdoor for specific application and user combination, effectively "removing encryption from the device of a terrorism suspect without weakening other devices in the market".
- Likewise, a targeted update to an operating system could introduce a backdoor to this specific operating system and user combination and thereby "facilitate access to information prior to or after an encryption method is employed".

[4] http://frankbraun.org/ref/assistance-and-access-bill-2018-explanatory-document.pdf

The motivation the Department of Home Affairs of the Australian Government gives for the Assistance and Access Bill 2018 is quite instructive:

> "Encryption conceals the content of communications and data held on devices, as well as the identity of users. Secure, encrypted communications are increasingly being used by terrorist groups and organised criminals to avoid detection and disruption. The problem is widespread, for example:
>
> - Encryption impacts at least nine out of every ten of ASIO's [Australian Security Intelligence Organisation] priority cases.
> - Over 90 per cent of data being lawfully intercepd by the AFP [Australian Federal Police] now use some form of encryption.
> - Effectively all communications among terrorists and organised crime groups are expected to be encrypted by 2020.
>
> State and Territory law enforcement are facing significant challenges as well. The following example from Victoria Police demonstrates:
>
> *A high risk Registered Sex Offender (RSO) was placed on the register for raping a 16 year old female, served nine years imprisonment and is now monitored by Corrections via two ankle bracelets whilst out on parole. Victoria Police received intelligence that he was breaching his RSO and parole conditions by contacting a number of females typically between 13 and 17 years of age. Enquiries showed that he was contacting these females and offering them drugs in return for sexual favours. The suspect was arrested and his mobile phone was seized but despite legislative requirements he refused to provide his passcode. Due to an inability to access his phone as well as the fact that he* ***used encrypted communication methods such as Snapchat and Facebook Messenger [sic]****, Victoria Police was unable to access evidence which would have enabled them to secure a successful prosecution and identify further victims and offences. These are high victim impact crimes that are being hindered by the inability of law enforcement to access encrypted communications.*
>
> Obstacles to the lawful access of communications significantly impacts the ability of law enforcement and security agencies to enforce the law, investigate serious crimes and protect the public. The measures in the Assistance and Access Bill will help our agencies overcome these challenges."

(from The Assistance and Access Bill 2018 website[5], accessed on 2018-10-04)

This approach to targeted backdoors won't be unique to Australia, they are just spearheading the approach for the Five Eye nations, and others are likely to follow.

[5] https://www.homeaffairs.gov.au/about/consultations/assistance-and-access-bill-2018

This document describes an approach to fight targeted backdoors with secure multiparty code reviews and a single source of truth (SSOT). This approach would render service providers **incapable** to comply with requests as described above, because it would ensure in an uncircumventable way that **all** users **always** get the same software update.

Current mitigation attempts

We now describe some of the currently deployed software signing methods.

Signing software with secure APT (Debian) packages

This is the model employed by `apt` in Debian and related distributions:

> "By adding a key to `apt`'s keyring, you're telling `apt` to trust everything signed by the key, and this lets you know for sure that apt won't install anything not *signed by the person who possesses the private key*."

(from SecureApt[6])

This leads to the reverse conclusion, that `apt` trusts everything signed by the person's private key.

`dpkg` has support for verifying GPG signatures of Debian package files, but this verification is disabled by default, only repository metadata is verified:

> GPG can be used to create a digital signature for both Debian package files and for APT repository metadata.
>
> Many Debian-based Linux distributions (e.g., Ubuntu) have GPG signature verification of Debian package files (.deb) disabled by default and instead choose to verify GPG signatures of repository metadata and source packages (.dsc). The setting which enables GPG signature checking of the individual .deb packages can be found in `/etc/dpkg/dpkg.cfg` and is set to `no-debsig`, but there are important caveats to enabling this option explained below.
>
> Further, most official Debian package files from the publicly accessible repositories *do not have* GPG signatures. The official repository metadata is GPG signed, as are the source packages, but the `.deb` packages themselves are not.
>
> If you publish a Debian package and GPG sign the package yourself before distributing it to users, those users' systems will, in most cases, t verify the signature of the package unless they have done a

[6] https://wiki.debian.org/SecureApt

> considerable amount of configuration. However, their system will, in most cases, automatically verify repository metadata.

(from HOWTO GPG sign and verify deb packages[7] and APT repositories)

Let's verify this on a fresh Ubuntu 18.04 x64 server instance (on DigitalOcean[8]):

```
root@Ubuntu:~# cat /etc/dpkg/dpkg.cfg
# dpkg configuration file
#
# This file can contain default options for dpkg.  All command-line
# options are allowed.  Values can be specified by putting them after
# the option, separated by whitespace and/or an `=' sign.
#

# Do not enable debsig-verify by default; since the distribution is not using
# embedded signatures, debsig-verify would reject all packages.
no-debsig

# Log status changes and actions to a file.
log /var/log/dpkg.log
```

And on a fresh Debian 9.5 x64 server instance (also on DigitalOcean):

```
root@Debian:~# cat /etc/dpkg/dpkg.cfg
# dpkg configuration file
#
# This file can contain default options for dpkg.  All command-line
# options are allowed.  Values can be specified by putting them after
# the option, separated by whitespace and/or an `=' sign.
#

# Do not enable debsig-verify by default; since the distribution is not using
# embedded signatures, debsig-verify would reject all packages.
no-debsig

# Log status changes and actions to a file.
log /var/log/dpkg.log
```

So there is indeed no package signing in Debian, only repository metadata signing with GPG.

GPG keys

Let's look at the GPG keys trusted for repository metadata signing.

[7] https://blog.packagecloud.io/eng/2014/10/28/howto-gpg-sign-verify-deb-packages-apt-repositories/

[8] https://www.digitalocean.com/

On Ubuntu 18.04 x64:

```
root@Ubuntu:~# apt-key list
/etc/apt/trusted.gpg.d/ubuntu-keyring-2012-archive.gpg
----------------------------------------------------
pub   rsa4096 2012-05-11 [SC]
      790B C727 7767 219C 42C8  6F93 3B4F E6AC C0B2 1F32
uid           [ unknown] Ubuntu Archive Automatic Signing Key (2012) <ftpmaster@ubuntu.com>

/etc/apt/trusted.gpg.d/ubuntu-keyring-2012-cdimage.gpg
----------------------------------------------------
pub   rsa4096 2012-05-11 [SC]
      8439 38DF 228D 22F7 B374  2BC0 D94A A3F0 EFE2 1092
uid           [ unknown] Ubuntu CD Image Automatic Signing Key (2012) <cdimage@ubuntu.com>
```

On Debian 9.5 x64:

```
root@Debian:~# apt-key list
/etc/apt/trusted.gpg.d/debian-archive-jessie-automatic.gpg
----------------------------------------------------------
pub   rsa4096 2014-11-21 [SC] [expires: 2022-11-19]
      126C 0D24 BD8A 2942 CC7D  F8AC 7638 D044 2B90 D010
uid           [ unknown] Debian Archive Automatic Signing Key (8/jessie) <ftpmaster@debian.

/etc/apt/trusted.gpg.d/debian-archive-jessie-security-automatic.gpg
-------------------------------------------------------------------
pub   rsa4096 2014-11-21 [SC] [expires: 2022-11-19]
      D211 6914 1CEC D440 F2EB  8DDA 9D6D 8F6B C857 C906
uid           [ unknown] Debian Security Archive Automatic Signing Key (8/jessie) <ftpmaste

/etc/apt/trusted.gpg.d/debian-archive-jessie-stable.gpg
-------------------------------------------------------
pub   rsa4096 2013-08-17 [SC] [expires: 2021-08-15]
      75DD C3C4 A499 F1A1 8CB5  F3C8 CBF8 D6FD 518E 17E1
uid           [ unknown] Jessie Stable Release Key <debian-release@lists.debian.org>

/etc/apt/trusted.gpg.d/debian-archive-stretch-automatic.gpg
----------------------------------------------------------
pub   rsa4096 2017-05-22 [SC] [expires: 2025-05-20]
      E1CF 20DD FFE4 B89E 8026  58F1 E0B1 1894 F66A EC98
uid           [ unknown] Debian Archive Automatic Signing Key (9/stretch) <ftpmaster@debian.
sub   rsa4096 2017-05-22 [S] [expires: 2025-05-20]

/etc/apt/trusted.gpg.d/debian-archive-stretch-security-automatic.gpg
--------------------------------------------------------------------
pub   rsa4096 2017-05-22 [SC] [expires: 2025-05-20]
      6ED6 F5CB 5FA6 FB2F 460A  E88E EDA0 D238 8AE2 2BA9
```

```
uid           [ unknown] Debian Security Archive Automatic Signing Key (9/stretch) <ftpmaste
sub   rsa4096 2017-05-22 [S] [expires: 2025-05-20]

/etc/apt/trusted.gpg.d/debian-archive-stretch-stable.gpg
--------------------------------------------------------
pub   rsa4096 2017-05-20 [SC] [expires: 2025-05-18]
      067E 3C45 6BAE 240A CEE8  8F6F EF0F 382A 1A7B 6500
uid           [ unknown] Debian Stable Release Key (9/stretch) <debian-release@lists.debian.

/etc/apt/trusted.gpg.d/debian-archive-wheezy-automatic.gpg
----------------------------------------------------------
pub   rsa4096 2012-04-27 [SC] [expires: 2020-04-25]
      A1BD 8E9D 78F7 FE5C 3E65  D8AF 8B48 AD62 4692 5553
uid           [ unknown] Debian Archive Automatic Signing Key (7.0/wheezy) <ftpmaster@debian

/etc/apt/trusted.gpg.d/debian-archive-wheezy-stable.gpg
-------------------------------------------------------
pub   rsa4096 2012-05-08 [SC] [expires: 2019-05-07]
      ED6D 6527 1AAC F0FF 15D1  2303 6FB2 A1C2 65FF B764
uid           [ unknown] Wheezy Stable Release Key <debian-release@lists.debian.org>
```

These keys are up to 6 years old and their possession is the only thing that is necessary to get clients to install updates.

Problems with this approach

In my opinion, the approach taken by APT has the following problems:

- Developer signatures do not matter for the end-user, only repository signatures.
- Repository signatures are created by a single GPG key which is often quite old and kept on a networked server, which seems to be the default method of running Debian repositories with Reprepro[9]. That means:
  - There is a single point of failure (just one key).
  - If keys are kept on a networked server they are easier to steal.
  - The long validity times make stolen keys more disastrous.
  - If the signing of packages is automatic then it is likely that there is little checking of package content happening.
- It doesn't contain a method for key rotation, in order to rotate keys packages are usually signed with "overlapping" keys for a while. This makes it hard to rotate keys in an emergency.
- It seems that packages have to be signed only by one trusted key in order to be accepted by an `apt` client. That is, there seems to be no pinned mapping between repositories and GPG keys. However, due to time constraints I

[9] https://wiki.debian.org/DebianRepository/SetupWithReprepro

wasn't able to verify this suspicion, please contact[10] me if you have more detailed information.

Signing software with signify in OpenBSD

OpenBSD[11] has one of the best deployed solutions for Securing OpenBSD From Us To You[12].

They implemented a simple minimal tool called `signify` to create and verify Ed25519 signatures and employed it to protect their base and package system. Keys are rotated with every release by putting the public keys for the next release in the current release.

Their simple setup and short keys makes it easy to verify the authenticity of keys, but they still suffer from most of the possible attacks described further below.

Signing software with the Git version control system

Git guarantees data integrity via Git's data structure (Merkle trees) and it allows to sign *tags* and *commits* with GPG.

Signing tags

The signing and verifying of tags in Git works as follows:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
$ git tag -v v1.5
```

The problem with signing tags is that they are not unmodifiable.

Signing commits

The signing and verifying of commits in Git works as follows:

```
$ git commit -a -S -m 'signed commit'
$ git merge --verify-signatures signed-branch
```

Only merging "fast-forwarding" branches gives some protection against regression (given one knows the `HEAD`).

The problem with signing commits is that *every* commit needs to be signed and the user has to trust *all* developer keys, which makes it hard to deploy in practice. And even then this model has no way of enforcing that at least two or

[10] mailto:frank@cryptogroup.net
[11] https://openbsd.org/
[12] https://www.openbsd.org/papers/bsdcan-signify.html

more developer sign each commit, allowing a single developer or compromised key to introduce backdoors.

Summary of possible attacks

The current solutions to sign software described above do not protect against the following possible attacks:

- Key compromise.
- Developer coercion (wrench attack), blackmailing, or bribing.
- Regression (suppression of updates).

A developer being forced to give up his signing key or a stolen repository signing key would be disastrous. Furthermore, GPG has no automatic mechanism for key rotation which is a likely reason why many GPG signing keys are quite old.

A proposed solution

We propose a design for *secure software distribution* and *secure software development* which mitigates the aforementioned possible attacks. In our implementation we get also get *verifiable build processes* for free, because the client compiles the source code himself and the source itself if fully secured. With a language that allows reproducible builds (like Go) this allows a verifiable build process.

The design tackles these to areas by:

1. Establishing code trust via multi-party code reviews recorded in unmodifiable hash chains. This prevents that a single developer can include a generic backdoor into software.
2. A single source of truth (SSOT) mechanism which makes sure every user of the software gets the same version of the software. This prevents targeted backdoors and the suppression of security updates.

Together this builds a secure software delivery and update mechanism which cannot be compromised by a single developer or for a specific user, thereby preventing *targeted backdoors*.

Below we describe the proposed design in more detail.

Code trust via multi-party code reviews recorded in hash chains

The general design looks like follows:

- The “unit” of code are directory trees.
- The hash of a directory tree is a *tree hash*.

- The *code history* is a sequence of *unique* tree hashes, starting from the hash of the empty tree.
- The sequence of tree hashes and *signatures* over them are recorded in a *hash chain* file.
- The signatures contributes towards a m-of-n threshold.
- Code is distributed as a set of patch files which transform a directory tree with hash **a** into a directory tree with hash **b**.
- Patch files are named after the outgoing tree hash **a**.

Tree hash specification

To calculate the hash of a directory tree (a *tree hash*) a list of all files in the directory root (a *tree list*) is created as follows.

All the files below the root of the directory tree are traversed in lexical order and printed in this format:

```
m xxx filename
```

Where:

```
m        is the mode ('f' or 'x')
xxx      is the SHA256 hash for the file in hex notation
filename is the file name with directory prefix starting at root
```

Example list:

```
f 7d865e959b2466918c9863afca942d0fb89d7c9ac0c99bafc3749504ded97730 bar/baz.txt
x b5bb9d8014a0f9b1d61e21e796d78dccdf1352f23cd32812f4850b878ae4944c foo.txt
```

The fields are separated with single white space characters and the lines are separated with single newline characters.

Directories are only implicitly listed (i.e., if they contain files). Entries start with 'f' if it is a regular file (read and write permission for user) and with 'x' if it is an executable (read, write, and executable for user).

The directory tree must only contain directories, regular files, or executables.

The deterministic tree list serves as the basis for a hash of a directory tree (the tree hash), which is the SHA256 hash of the tree list in hex notation.

Hash chain file format

A hash chain is a chain of signatures over a chain of code changes.

A hash chain is stored in a simple newline separated text file where each hash chain entry corresponds to a single line and has the following form:

```
hash-of-previous current-time type type-fields ...
```

Where hash-of-previous is the SHA256 hash of the previous line (without the trailing newline) in hex encoding. The fields are separated by single white spaces. The current-time is encoded as an ISO 8601 string in UTC.

All hashes in a hash chain are SHA256 hashes encoded in hex notation. Hex encodings have to be lowercase. All public keys are Ed25519 keys and they and their signatures are encoded in base64 (URL encoding without padding). Comments are arbitrary UTF-8 sequences, but cannot contain newlines.

There are six different types of hash chain entries:

```
cstart
source
signtr
addkey
remkey
sigctl
```

A hash chain must start with a cstart entry and that is the only line where this type must appear.

Type cstart

A cstart entry starts a new hash chain.

```
hash-of-previous current-time cstart pubkey nonce signature [comment]
```

The hash-of-previous for the cstart time is the hash of an empty source tree. The signature by pubkey is over the pubkey, the nonce, and the optional comment. The comment should identify the owner of the pubkey, not the project. The nonce must be a 24 byte random number in base64 (URL encoding without padding). This makes pubkey the only valid signer for the hash chain and implicitly sets the signature threshold m to 1.

Type source

A source entry marks a new source tree state for publication from the developer owning the signing pubkey. The optional comment can be used to describe the change to the reviewers.

```
hash-of-previous current-time source tree-hash pubkey signature [comment]
```

The signature by pubkey is over the source tree hash and the optional comment. See the tree package for a detailed description of source tree hashes.

Type signtr

A signtr entry signs a previous hash chain entry and thereby approves all code changes and changes to the set of signature keys and m up to that point.

```
hash-of-previous current-time signtr hash-of-chain-entry pubkey signature
```

It does not necessarily sign the previous line and can therefore be done in a detached fashion by a reviewer and added later by the developer responsible for maintaining the hash chain. This avoids merge conflicts.

Type addkey

An addkey entry marks a signature pubkey for addition to the list of approved signature keys.

```
hash-of-previous current-time addkey w pubkey signature [comment]
```

The weight of the key towards the minimum number of necessary signatures m is denoted by w. The pubkey can be accompanied by an optional comment, but the signature must be over both. The comment is added last so it can contain white spaces without complicating the parsing, it should identify the owner of the pubkey.

Type remkey

A remkey entry marks a signature pubkey for removal from the list of approved signature keys.

```
hash-of-previous current-time remkey pubkey
```

Type sigctl

A sigctl entry denotes an update of m, the minimum number of necessary signatures to approve state changes (the threshold).

```
hash-of-previous current-time sigctl m
```

Patchfile format

A patchfile is a UTF-8 encoded file split into newline separated lines. It starts with the following line which defines the patchfile version:

```
codechain patchfile version 1
```

The second line gives the tree hash of the directory tree the patchfile applies to (example):

```
treehash 5998c63aca42e471297c0fa353538a93d4d4cfafe9a672df6989e694188b4a92
```

The main body of the patch file encodes file deletions, file additions, and file diffs. A file deletion is encoded as follows (example):

```
- f 927d2cae58bb53cdd087bb7178afeff9dab8ec1691cbd01aeccae62559da2791 gopher.png
```

The '-' denotes a deletion. The other three entries are the same as file entries of tree lists.

A file addition is encoded as follows (example):

```
+ f ad125cc5c1fb680be130908a0838ca2235db04285bcdd29e8e25087927e7dd0d hello.go
```

The '+' denotes an addition. The other three entries are the same as file entries of tree lists.

After an addition the actual patch must follow, either in "dmppatch" (for UTF-8 files) or in "ascii85" format (for binary files). The "dmppatch" file format looks like the following (example):

```
dmppatch 2
@@ -0,0 +1,78 @@
+package main%0A%0Aimport (%0A%09%22fmt%22%0A)%0A%0Afunc main() %7B%0A%09fmt.Println(%22hell
```

The number after "dmppatch" denotes the number of lines following containing the actual UTF-8 patch.

The "ascii85" file format looks like the following (example):

```
ascii85 2
+,^C)8Mp-E!!DW60b/e#'ElcGar]O1ZH.;>ZnWJO:iLd/`5G7uXPR`iQmqOB\]npD=)8AK4gPQFI-+W_
>oidmeIj`.fgNufo<4MB5*&XfkqnCOo9\::*WQO?z!!*#^!R=9-%KImW!!
```

The number after "ascii85" denotes the number of lines following containing the actual binary encoding. "ascii85" patches are not real patches, but always encode the entire binary file.

A file diff is encoded as follows (example):

```
- f ad125cc5c1fb680be130908a0838ca2235db04285bcdd29e8e25087927e7dd0d hello.go
+ f 1b239e494fa201667627de82f0e4dc27b7b00b6ec06146e4d062730bf3762141 hello.go
```

As with file additions, after a file diff the actual patch must follow, either in "dmppatch" or "ascii85" format (see above), if the file hash changed. That is, if just the file mode changed and the file hash stayed the same no patch must follow.

File diffs are only used if the file names ("hello.go" in the example above) are the same. File moves are implemented as a file deletion and a file addition.

The last line in a patchfile must be the tree hash of the directory tree after the patchfile has been applied (example):

```
treehash e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
```

Patchfiles are optimized for robustness, not for compactness or human readability (although the human readability is reasonable). A complete example containing a single UTF-8 file addition:

```
codechain patchfile version 1
treehash e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
+ f ad125cc5c1fb680be130908a0838ca2235db04285bcdd29e8e25087927e7dd0d hello.go
dmppatch 2
@@ -0,0 +1,78 @@
+package main%0A%0Aimport (%0A%09%22fmt%22%0A)%0A%0Afunc main() %7B%0A%09fmt.Println(%22hell
treehash 5998c63aca42e471297c0fa353538a93d4d4cfafe9a672df6989e694188b4a92
```

Diff function specification

Given the patchfile format described above, a Diff function that computes a patch file for two directory trees rooted at A and B is straightforward to implement:

1. Calculate tree lists LIST_A and LIST_B (in lexical order) for A and B.
2. Compare the file names NAME_A and NAME_B (lexicographically) of the first two entries in LIST_A and LIST_B:
   - If NAME_A < NAME_B: File delete NAME_A, remove it from LIST_A, goto 2.
   - If NAME_A > NAME_B: File add NAME_B, remove it from LIST_B, goto 2.
   - If NAME_A == NAME_B:
   - If file mode or file hash of files NAME_A and NAME_B differ: file diff.
   - Remove NAME_A from LIST_A, NAME_B from LIST_B, and goto 2.
3. If LIST_A still contains entries while LIST_B is empty, add file deletions for all entries in LIST_A.
4. If LIST_B still contains entries while LIST_A is empty, add file additions for all entries in LIST_B.

Apply function specification

To apply a patchfile PATCH to a directory DIR we use the following algorithm:

1. Read first line of of PATCH and make sure it contains a codechain patchfile version we understand.
2. Read the second line of PATCH, make sure it is a treehash, and compare it with the treehash of DIR (before any patches have been applied).
3. Read next line of PATCH:

- If it starts with '+': Add file encoded in the following patch.
- If it starts with '-':
- If the next line starts with '+':
  - If the file name differ: Delete first file, add second file (with the following patch, which must be either ascii85 or dmppatch).
  - Otherwise (file names are the same):
  - If hashes are the same (only file modes differ): Adjust mode.
  - Otherwise (hashes differ): Apply the following patch, which must be either ascii85 or dmppatch (and adjust mode, if necessary).
- Otherwise: Delete file.
- If it starts with 'treehash': Goto 4.
- Goto 3.

4. Read the last line of PATCH, make sure it is a treehash, and compare it with the treehash of DIR (after all patches have been applied).

Distributing the current head

The current head of a hash chain is all one need to fully verify the entire code history and recreate the most current code version with enough signatures, given that one has access to the hash chain and the corresponding patch files.

But in order to prevent the suppression of updates to certain users, a form of targeted updates, one has to ensure that all users have access to the most current head.

The method to do that is employing a so-called single source of truth (SSOT) where every user has access to the same authentic version of a data object. See also SSOT on Wikipedia[13].

Single source of truth (SSOT) via DNS

One widely deployed SSOT system which can be used for that is the Domain Name System (DNS) which associates various information with domain names assigned to each of the participating entities (see DNS on Wikipedia[14]).

In our design we store the necessary information, a *signed head*, in the TXT record[15] of a fully qualified domain name (FQDN[16]) starting with the prefix `_codechain.`

The head is signed, which allows clients to verify updates to it, after they have seen it for the first time and learning the corresponding public key (trust on first use).

[13] https://en.wikipedia.org/wiki/Single_source_of_truth
[14] https://en.wikipedia.org/wiki/Domain_Name_System
[15] https://en.wikipedia.org/wiki/TXT_record
[16] https://en.wikipedia.org/wiki/Fully_qualified_domain_name

Due to the distributed caching design of DNS it is not possible for publishers to send different signed heads to different users, which prevents *targeted updates* by publishers.

Distributing false signed heads through DNS spoofing[17] is prevented **if and only if** the client has seen a signed head before (clients cache public keys)

If a client has not seen a valid head before it is vulnerable to DNS spoofing. However, this can be mitigated by deploying the SSOT on a domain which is secured by the Domain Name System Security Extensions (DNSSEC[18]).

In order to publish packages using a single source of truth (SSOT) with DNS TXT records we first define how the *signed heads* are represented as TXT records and then define two noperations: "CreatePkg" to initially publish a package and "SignHead" to publish updates.

Signed head specification

Signed heads have the following fields:

- `PUBKEY` (32-byte), the Ed25519 public key of SSOT head signer.
- `PUBKEY_ROTATE` (32-byte), Ed25519 pubkey to rotate to, set to 0 if unused.
- `VALID_FROM` (8-byte), the signed head is valid from the given Unix time.
- `VALID_TO` (8-byte), the signed head is valid to the given Unix time.
- `COUNTER` (8-byte), strictly increasing signature counter.
- `HEAD`, the Codechain head to sign.
- `SIGNATURE`, signature with `PUBKEY`.

The `SIGNATURE` is over all previous fields:

`PUBKEY|PUBKEY_ROTATE|VALID_FROM|VALID_TO|COUNTER|HEAD`

The signed head is a concatenation of

`PUBKEY|PUBKEY_ROTATE|VALID_FROM|VALID_TO|COUNTER|HEAD|SIGNATURE`

encoded in base64 (URL encoding without padding.

All integers (`VALID_FROM`, `VALID_TO`, `COUNTER`) are encoded in network order (big-endian).

CreatePkg specification

To create a new secure package for a project developed with Codechain that should be distributed with a SSOT using DNS TXT records, the following procedure is defined:

1. Make sure the project with NAME has not been published before. That is, the directory `~/.config/ssotpub/pkgs/NAME` does not exist.

[17] https://en.wikipedia.org/wiki/DNS_spoofing
[18] https://en.wikipedia.org/wiki/Domain_Name_System_Security_Extensions

2. Create a new `.secpkg` file which specifies the following:

   - The NAME of the project.
   - The fully qualified domain name (DNS) where the TXT records can be queried.
   - The URL under which the distribution .tar.gz files can be downloaded.
   - The current HEAD of the project's Codechain.

   The .secpkg file is saved to the current working directory, which is typically added to the root of the project's repository.

3. Create the first signed head for the current project's HEAD with a supplied secret key and counter set to 0.

4. Create the directory `~/.config/ssotpub/pkgs/NAME/dists` and save the current distribution to `~/.config/ssotpub/pkgs/NAME/dists/HEAD.tar.gz` (`codechain createdist`).

5. Save the signed head to `~/.config/ssotpub/pkgs/NAME/signed_head`

6. Print the distribution name:
   `~/.config/secpkg/pkgs/NAME/dists/HEAD.tar.gz`

7. Print DNS TXT record as defined by the .secpkg and the first signed head.

Afterwards the administrator manually uploads the distribution HEAD.tar.gz to the download URL and publishes the new DNS TXT record in the defined zone. DNSSEC should be enabled.

SignHead specification

To publish an update of a secure package with SSOT do the following:

1. Parse the .secpkg file in the current working directory.

2. Make sure the project with NAME has been published before. That is, the directory `~/.config/ssotpub/pkgs/NAME` exists.

3. Validate the signed head in `~/.config/ssotpub/pkgs/NAME/signed_head` and make sure the corresponding secret key is available.

4. Get the HEAD from `.codechain/hashchain` in the current working directory.

5. Create a new signed head with current HEAD, the counter of the previous signed head plus 1, and update the saved signed head:

   - Copy `~/.config/ssotpub/pkgs/NAME/signed_head` to `~/.config/ssotpub/pkgs/NAME/previous_signed_head`
   - Save new signed head to `~/.config/ssotpub/pkgs/NAME/signed_head` (atomic).

6. Save the current distribution to:
   `~/.config/secpkg/pkgs/NAME/dists/HEAD.tar.gz`
   (`codechain createdist`).
7. Print the distribution name:
   `~/.config/ssotpkg/pkgs/NAME/dists/HEAD.tar.gz`
8. Print DNS TXT record as defined by the `.secpkg` and the signed head.
9. If the HEAD changed, update the `.secpkg` file accordingly.

Afterwards the administrator manually uploads the distribution HEAD.tar.gz to the download URL and publishes the new DNS TXT record in the defined zone. DNSSEC should be enabled.

Secure package (`.secpkg`) specification

A secure package (`.secpkg` file) contains a JSON object with the following keys:

```
{
  "Name": "the project's package name",
  "Head": "head of project's Codechain",
  "DNS": "fully qualified domain name",
  "URL": "URL to download project files of the from (URL/head.tar.gz)"
}
```

Example `.secpkg` file for Codechain itself:

```
{
  "Name": "codechain",
  "Head": "73fe1313fd924854f149021e969546bce6052eca0c22b2b91245cb448410493c",
  "DNS": "codechain.secpkg.net",
  "URL": "http://frankbraun.org/codechain"
}
```

Install specification

Installing software described by a `.secpkg` file works as follows:

1. Parse .secpkg file and validate it. Save head as HEAD_PKG.
2. Make sure the project with NAME has not been installed before. That is, the directory `~/.config/secpkg/pkgs/NAME` does not exist.
3. Create directory `~/.config/secpkg/pkgs/NAME`
4. Save `.secpkg` file to `~/.config/secpkg/pkgs/NAME/.secpkg`
5. Query TXT record from _codechain.DNS and validate the signed head contained in it. Save head from TXT record (HEAD_SSOT).

6. Store the signed head to `~/.config/secpkg/pkgs/NAME/signed_head`
7. Download distribution file from URL/HEAD_SSOT.tar.gz and save it to `~/.config/secpkg/pkgs/NAME/dists`
8. Apply `~/.config/secpkg/pkgs/NAME/dists/HEAD_SSOT.tar.gz` to `~/.config/secpkg/pkgs/NAME/src` with `codechain apply -f ~/.config/secpkg/pkgs/NAME/dists/HEAD_SSOT.tar.gz -head HEAD_SSOT`
9. Make sure HEAD_PKG is contained in `~/.config/secpkg/pkgs/NAME/src/.codchain/hashchain`
10. Copy `~/.config/secpkg/pkgs/NAME/src` to `~/.config/secpkg/pkgs/NAME/build`
11. Call `make prefix=~/.config/secpkg/local` in `~/.config/secpkg/pkgs/NAME/build`
12. Call `make prefix= ~/.config/secpkg/local install` in `~/.config/secpkg/pkgs/NAME/build`
13. Move `~/.config/secpkg/pkgs/NAME/build` to `~/.config/secpkg/pkgs/NAME/installed`

If the installation process fails at any stage during the procedure described above, report the error and remove the directory `~/.config/secpkg/pkgs/NAME`.

For the process above to work, the projects distributed as secure packages must contain a Makefile (for GNU Make) with the "all" target building the software and the "install" target installing it.

The software must be self-contained without any external dependencies, except for the compiler. For Go software that means at least Go 1.11 must be installed (with module support) and all dependencies must be vendored.

Update specification

Updating a software package with NAME works as follows:

1. Make sure the project with NAME has been installed before. That is, the directory `~/.config/secpkg/pkgs/NAME` exists.
2. Load .secpkg file from `~/.config/secpkg/pkgs/NAME/.secpkg`
3. Load signed head from `~/.config/secpkg/pkgs/NAME/signed_head` (as DISK)
4. Query TXT record from _codechain.DNS, if it is the same as DISK, goto 15.
5. Validate signed head from TXT and store HEAD:

- pubKey from TXT must be the same as pubKey or pubKeyRotate from DISK.
- The counter from TXT must be larger than the counter from DISK.
- The signed head must be valid (as defined by validFrom and validTo).

If the validation fails, abort update procedure and report error.

6. If signed head from TXT record is the same as the one from DISK:
   - Copy `~/.config/secpkg/pkgs/NAME/signed_head` to `~/.config/secpkg/pkgs/NAME/previous_signed_head`
   - Save new signed head to `~/.config/secpkg/pkgs/NAME/signed_head` (atomic).
   - Goto 15.
7. Download distribution file from URL/HEAD.tar.gz and save it to `~/.config/secpkg/pkgs/NAME/dists`
8. Apply `~/.config/secpkg/pkgs/NAME/dists/HEAD.tar.gz` to `~/.config/secpkg/pkgs/NAME/src` with `codechain apply -f ~/.config/secpkg/pkgs/NAME/dists/HEAD.tar.gz -head HEAD`.
9. `rm -rf ~/.config/secpkg/pkgs/NAME/build`
10. `cp -r ~/.config/secpkg/pkgs/NAME/src ~/.config/secpkg/pkgs/NAME/build`
11. Call `make prefix=~/.config/secpkg/local` in `~/.config/secpkg/pkgs/NAME/build`
12. Call `make prefix= ~/.config/secpkg/local install` in `~/.config/secpkg/pkgs/NAME/build`
13. `mv ~/.config/secpkg/pkgs/NAME/build ~/.config/secpkg/pkgs/NAME/installed`
14. Update signed head:
    - Copy `~/.config/secpkg/pkgs/NAME/signed_head` to `~/.config/secpkg/pkgs/NAME/previous_signed_head`
    - Save new signed head to `~/.config/secpkg/pkgs/NAME/signed_head` (atomic).
15. The software has been successfully updated.

Implementation in Codechain

The specifications given above are implemented in Codechain[19] (in Go). Codechain contains three tools for different user roles:

[19] https://github.com/frankbraun/codechain

1. `codechain` for developers to record code changes and corresponding multi-party reviews in a unmodifiable hash chain.
2. `ssotpub` for admins to publish the head of a hash chain created by `codechain` with a SSOT using DNS TXT records, creating a `.secpkg` file in the process.
3. `secpkg` for users to securely install and update software distributed as `.secpkg` files.

Codechain goals

- Signed multiparty code reviews.
- Easy & built-in key rotation.
- Protection against $5 wrench attack.
- Regression protection, unmodifiable history.
- Minimal usable implementation written in Go as soon as possible.
- Focus on **source** distribution, not binary.
- Single source of truth (SSOT).

Out-of-scope:

- Source code management (just use Git).
- Code distribution (minimal support is provided via `codechain createdist` and `codechain apply -f`, both are used by `secpkg`).

Tree hash implementation

In Codechain tree hashes and tree lists are implemented in the tree package[20] and available through the `codechain treehash` command.

Example use:

```
$ cd $GOPATH/src/github.com/frankbraun/codechain/doc/helloproject

$ codechain treehash -l
f ab81f3080f71a034c90dc0ca64b62295d3a75a23ec1b0f498dfda4a34325ae3a README.md
f ad125cc5c1fb680be130908a0838ca2235db04285bcdd29e8e25087927e7dd0d hello.go

$ codechain treehash
d844cbe6f6c2c29e97742b272096407e4d92e6ac7f167216b321c7aa55629716

$ codechain treehash -l | sha256sum
d844cbe6f6c2c29e97742b272096407e4d92e6ac7f167216b321c7aa55629716
```

[20] https://godoc.org/github.com/frankbraun/codechain/tree

Hash chain implementation

The hash chain file format is implemented in the hashchain package[21], look there for an example hash chain.

Patchfile implementation

The patchfile format is implemented in the patchfile package[22].

SSOT via DNS implementation

SSOT with DNS TXT records is implemented in the SSOT package[23].

Secure package implementation

Secure packages are implemented in the secpkg package[24].

Conclusion

The design for secure software development and distribution with secure multi-party code reviews and a SSOT described in this document and implemented in Codechain gives us

- globally identical,
- verifiable,
- reproducible, and
- attributable

binaries build from source, which is a *technical mitigation* against one of the most likely results of further crypto regulation attempts, so-called *targeted updates*.

We built a minimal solution in order to be able to use it for our own software, but I hope that others will build upon it and improve the design and/or implementation to close what I believe is a gaping security hole in the area of software updates.

Future work: platform vendors

While many open-source software distributions came a long way towards *verifiable build processes* (see Reproducible Builds[25]), they often lack in the area of *secure software distribution* and *secure software development*. The most common signing mechanism used by open-source software projects and distributions nowadays—a

[21] https://godoc.org/github.com/frankbraun/codechain/hashchain
[22] https://godoc.org/github.com/frankbraun/codechain/patchfile
[23] https://godoc.org/github.com/frankbraun/codechain/ssot
[24] https://godoc.org/github.com/frankbraun/codechain/secpkg
[25] https://reproducible-builds.org/

single GPG signature—is not sufficient for secure software development. And we are not aware of any secure software distribution mechanism employed by distributions today that guarantees that the installed software is the same as on all devices.

We believe a design like the one described in this document and implementations like Codechain[26] could be used to improve that situation for open-source distributions, make targeted backdoors impossible there, and by advancing the state-of-the-art put pressure on commercial vendors to implement similar mechanisms.

In my opinion, the following open-source operating systems are the best candidates: Arch Linux[27], Gentoo Linux[28], and OpenBSD[29].

For commercial OS vendors especially Apple, with their newfound focus on privacy and security that is aligned with their core business, seems to be a viable candidate.

Acknowledgments

Codechain and this paper have been heavily influenced by discussions with Jonathan Logan[30] of Cryptohippie[31], Inc.

[26] https://github.com/frankbraun/codechain
[27] https://archlinux.org/
[28] https://gentoo.org/
[29] https://openbsd.org/
[30] https://github.com/JonathanLogan
[31] https://secure.cryptohippie.com/