In code we trust: Fighting targeted backdoors with secure multiparty code reviews and a single source of truth

Frank Braun

@thefrankbraun

2018-10-06

who am I?

- cryptoanarchist
- software developer

⇒ **but**: I became somewhat disillusioned with mainstream tech...

- talk last year on "Dehumanizing Technology":
- IMHO current trajectory leads to technological totalitarianism

what do?

- political reform (→ good luck!)
- abolish ("bombing us back into the stone age"[1])
- transform (grow faster on a different technology trajectory)

⇒ ~~cryptoanarchistic~~ software engineering (freedom technology)

[1] Ted Kaczynski, *Anti-Tech Revolution: Why and How* (2016)

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, 1984*

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, 1984*

questions:

- how can we trust the people who wrote the software?

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, 1984*

questions:

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

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, 1984*

questions:

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

$\Rightarrow$ this talk is not about making sure the code you execute is right,

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, 1984*

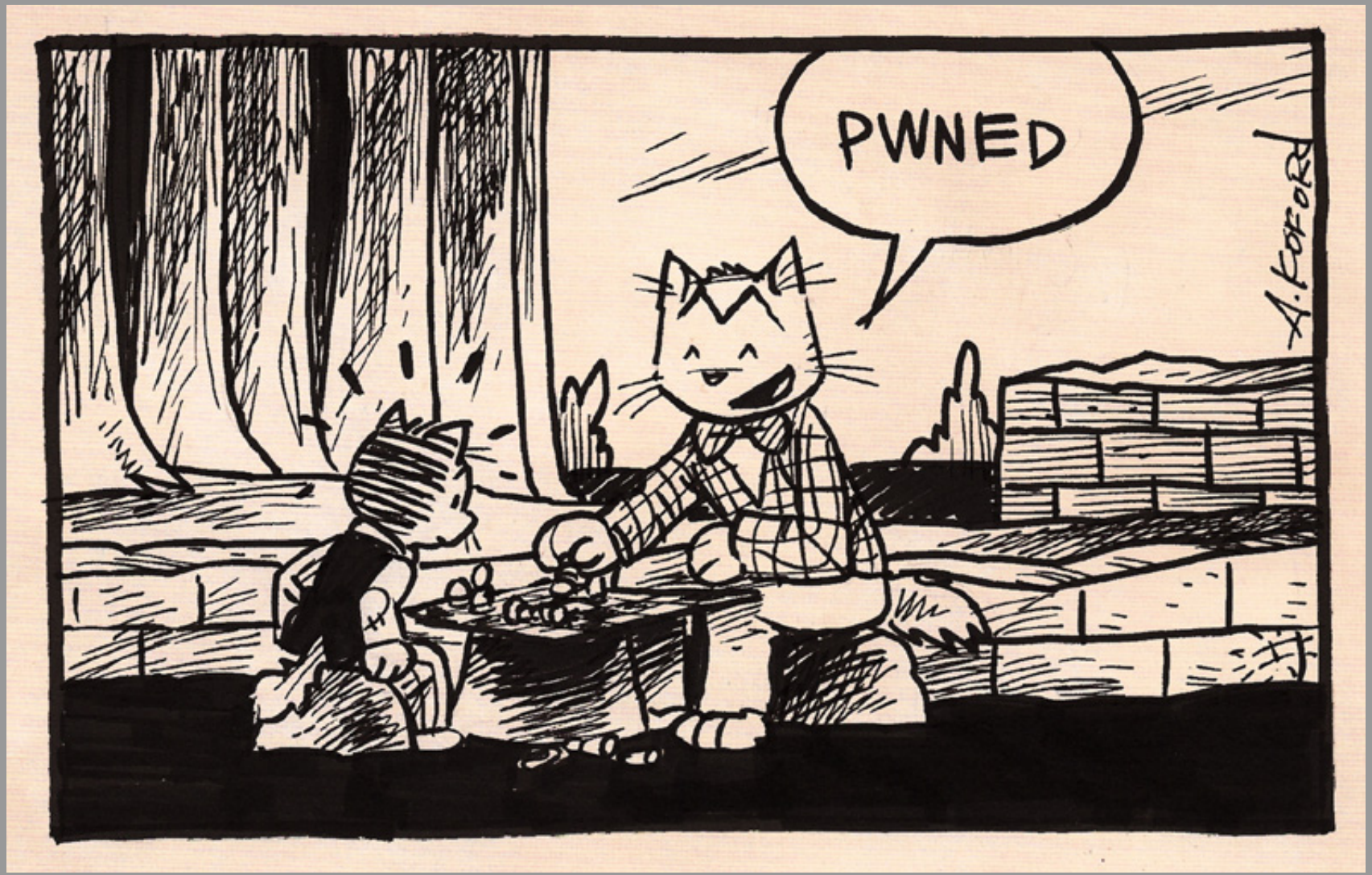
questions:

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

$\Rightarrow$ this talk is not about making sure the code you execute is right, but making sure you execute the right code!

what happens if you execute the wrong code?

what happens if you execute the wrong code?



the problem of targeted backdoors

The Fog of Cryptowar[2] warned:

> *"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."*

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

the problem of targeted backdoors

The Fog of Cryptowar[2] warned:

> *"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."*

$\Rightarrow$ crypto regulation **not** black-and-white, not about outlawing

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

realistic and likely regulatory approaches

realistic and likely regulatory approaches

1. defense of metadata access

realistic and likely regulatory approaches

1. defense of metadata access
2. nudge vendors to deliver software with less secure defaults

realistic and likely regulatory approaches

1. defense of metadata access
2. nudge vendors to deliver software with less secure defaults
3. lawful hacking

realistic and likely regulatory approaches

1. defense of metadata access
2. nudge vendors to deliver software with less secure defaults
3. lawful hacking
4. use of update mechanisms to deliver police Trojans

realistic and likely regulatory approaches

1. defense of metadata access
2. nudge vendors to deliver software with less secure defaults
3. lawful hacking
4. use of update mechanisms to deliver police Trojans
5. mandate plaintext access

realistic and likely regulatory approaches

1. defense of metadata access
2. nudge vendors to deliver software with less secure defaults
3. lawful hacking
4. use of update mechanisms to deliver police Trojans
5. mandate plaintext access

focus on technical mitigations to issue 4, the use of targeted updates to introduce backdoors into specific devices to surveil the user (so-called targeted backdoors)

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.
2. Secure software development: Better review and auditing for security critical code. Enforce that code changes always require at least two signatures.

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.
2. Secure software development: Better review and auditing for security critical code. Enforce that code changes always require at least two signatures.
3. Verifiable build processes: Review and audits are of little use, if the build process (the compilation) cannot be verified.

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.
2. Secure software development: Better review and auditing for security critical code. Enforce that code changes always require at least two signatures.
3. Verifiable build processes: Review and audits are of little use, if the build process (the compilation) cannot be verified.
4. More decentralized platform vendors: Today very few platform providers control the OS and the app delivery channels.

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.
2. Secure software development: Better review and auditing for security critical code. Enforce that code changes always require at least two signatures.
3. Verifiable build processes: Review and audits are of little use, if the build process (the compilation) cannot be verified.
4. ~~More decentralized platform vendors: Today very few platform providers control the OS and the app delivery channels.~~

relevant suggestions

1. Secure software distribution: Automatically verify with of single source of truth (SSOT) that a program is the same on all devices.
2. Secure software development: Better review and auditing for security critical code. Enforce that code changes always require at least two signatures.
3. ~~Verifiable build processes: Review and audits are of little use, if the build process (the compilation) cannot be verified.~~
4. ~~More decentralized platform vendors: Today very few platform providers control the OS and the app delivery channels.~~

recent developments regarding targeted updates

Unfortunately, that's exactly the approach the Australian government, a member of the Five Eyes, is taking in their proposed Assistance and Access Bill 2018:

recent developments regarding targeted updates

Unfortunately, that's 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 systematically weaken these products across the market."*

recent developments regarding targeted updates

Unfortunately, that's 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 systematically weaken these products across the market."*

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

Assistance and Access Bill 2018

> *"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."*

Assistance and Access Bill 2018

> *"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."*

$\Rightarrow$ targeted updates are the available technical methods which allow to give governments these targeted backdoors!

secure APT (Debian) packages

that's the model employed by apt in Debian and related distros

> *"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."*—`https://wiki.debian.org/SecureApt`

secure APT (Debian) packages

that's the model employed by apt in Debian and related distros

> *"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."*—`https://wiki.debian.org/SecureApt`

reverse conclusion:
apt trusts everything signed by the person's private key

secure APT (Debian) packages

that's the model employed by apt in Debian and related distros

> *"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."*—`https://wiki.debian.org/SecureApt`

reverse conclusion:
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!
- Debian & Ubuntu: GPG keys are up to 6 years old

problem with APT's approach

- Developer signatures do not matter, only repository sigs.
- Repo sigs created by single GPG key, often quite old, and kept on networked server:
  - 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.
- No method for key rotation, to rotate keys packages are signed with "overlapping" keys $\Rightarrow$ hard to rotate in 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, no pinned mapping between repositories and GPG keys?

Git version control system

- data integrity via Git's data structure (Merkle trees)
- Git allows to sign tags and commits with GPG

Git: signing & verifying tags

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

Git: signing & verifying tags

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

problem:

- tags are not unmodifiable

Git: signing & verifying commits

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

Git: signing & verifying commits

```
$ 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`)

Git: signing & verifying commits

```
$ 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`)

problem:

- every commit needs to be signed
- user has to trust all developer keys

Git: signing & verifying commits

```
$ 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`)

problem:

- every commit needs to be signed
- user has to trust all developer keys

$\Rightarrow$ hard to deploy in practice

summary of possible attacks

The current solutions to sign software do not protect against:

summary of possible attacks

The current solutions to sign software do not protect against:

- key compromise

summary of possible attacks

The current solutions to sign software do not protect against:

- key compromise
- developer coercion (wrench attack), blackmailing, or bribing

summary of possible attacks

The current solutions to sign software do not protect against:

- key compromise
- developer coercion (wrench attack), blackmailing, or bribing
- regression (suppressing of updates)

summary of possible attacks

The current solutions to sign software do not protect against:

- key compromise
- developer coercion (wrench attack), blackmailing, or bribing
- regression (suppressing of updates)

A developer being forced to give up his signing key or a stolen repository signing key would be disastrous.

summary of possible attacks

The current solutions to sign software do not protect against:

- key compromise
- developer coercion (wrench attack), blackmailing, or bribing
- regression (suppressing of updates)

A developer being forced to give up his signing key or a stolen repository signing key would be disastrous.

GPG has no automatic mechanism for key rotation, likely reason why many GPG keys are quite old.

proposed solution

design for secure software distribution and development which mitigates these attacks:

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.

Codechain design

(joined work with Jonathan "Smuggler" Logan)

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- the "unit" of code are directory trees

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- the "unit" of code are directory trees
- the hash of a directory tree is a tree hash

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- 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

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- 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 their signatures are recorded in a hash chain file

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- 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 their signatures are recorded in a hash chain file
- the signatures contributes towards a m-of-n threshold

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- 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 their signatures 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 *a* into a directory tree *b*

Codechain design

(joined work with Jonathan "Smuggler" Logan)

- 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 their signatures 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 *a* into a directory tree *b*
- patch files are named after the outgoing tree hash *a*

tree hashes

```
$ 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
```

patch files

```
codechain patchfile version 1
treehash e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495
+ f ab81f3080f71a034c90dc0ca64b62295d3a75a23ec1b0f498dfda4a34
dmppatch 2
@@ -0,0 +1,45 @@
+## Example project for Codechain walkthrough%0A
+ f ad125cc5c1fb680be130908a0838ca2235db04285bcdd29e8e2508792
dmppatch 2
@@ -0,0 +1,78 @@
+package main%0A%0Aimport (%0A%09%22fmt%22%0A)%0A%0Afunc main
%7B%0A%09fmt.Println(%22hello world!%22)%0A%7D%0A
treehash d844cbe6f6c2c29e97742b272096407e4d92e6ac7f167216b321
```

hash chain format

- a hash chain is stored in a simple newline separated text file

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)
- the fields are separated by single white spaces

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

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

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)
- 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

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)
- 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

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)
- 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)

hash chain format

- a hash chain is stored in a simple newline separated text file
- each hash chain entry corresponds to a single line of the form:

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

where:

- hash-of-previous is the SHA256 hash of the previous line (without newline)
- 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 (without newlines)

hash chain types

there are six different types of hash chain entries:

hash chain types

there are six different types of hash chain entries:

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

hash chain types

there are six different types of hash chain entries:

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

- a hash chain must start with a `cstart` entry
- that is the only line where this type must appear

type `cstart`

A `cstart` entry starts a new hash chain.

type `cstart`

A `cstart` entry starts a new hash chain.

```
hash-of-prev cur-time cstart pubkey nonce signature [comment]
```

`type source`

Marks a new source tree state for publication (from developer).

type source

Marks a new source tree state for publication (from developer).

```
hash-of-prev cur-time source source-hash pubkey sig [comment]
```

Signature is over the source-hash and the optional comment.

type `signtr`

Signs a previous entry and approves all code changes and changes to the set of signature keys and `m` up to that point.

type `signtr`

Signs a previous entry and approves all code changes and changes to the set of signature keys and `m` up to that point.

```
hash-of-prev cur-time signtr hash-of-chain-entry pubkey sig
```

It does not have to sign the previous line ($\rightarrow$ detached signatures).

type addkey

Marks a pubkey for addition to the list of approved signature keys.

type addkey

Marks a pubkey for addition to the list of approved signature keys.

```
hash-of-prev cur-time addkey w pubkey sig [comment]
```

The weight (towards `m`) is denoted by `w`.

type `remkey`

A `remkey` entry marks a signature pubkey for removal.

type `remkey`

A `remkey` entry marks a signature pubkey for removal.

```
hash-of-prev cur-time remkey pubkey
```

type `sigctl`

Denotes an update of `m`, the minimum number of necessary signatures to approve state changes (the threshold).

type `sigctl`

Denotes an update of `m`, the minimum number of necessary signatures to approve state changes (the threshold).

```
hash-of-prev cur-time sigctl m
```

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.

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.

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.

$\Rightarrow$ Employ a so-called single source of truth (SSOT) where every user has access to the same authentic version of a data object.

Single source of truth (SSOT) via DNS

- use widely deployed SSOT system DNS
- store a signed head in TXT record of fully qualified domain name
- the head is signed, allows clients updates to it (trust on first use)
- due to distributed caching of DNS it is not possible for publishers to send different signed heads to different users (no targeted updates by publishers)
- distributing false signed heads through DNS spoofing is prevented **iff** client has seen signed head before (clients cache)
- if valid head not seen before, vulnerable to DNS spoofing
- can be mitigated by deploying the SSOT on a domain with DNSSEC

Signed head specification

- `PUBKEY` (32-byte), Ed25519 public key of SSOT head signer
- `PUBKEY_ROTATE` (32-byte), pubkey to rotate to (0 if unused)
- `VALID_FROM` (8-byte), signed head valid from given Unix time
- `VALID_TO` (8-byte), 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`

concatenate (integers in big-endian) and encode as base64

Secure package (`.secpkg`) specification

```
{
  "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"
}
```

Example '.secpkg' file for Codechain itself

```
{
  "Name": "codechain",
  "Head": "73fe1313fd924854f149021e969546bce6052eca0c22b2
  "DNS": "codechain.secpkg.net",
  "URL": "http://frankbraun.org/codechain"
}
```

CreatePkg & SignHead

- CreatePkg: Publish HEAD & distribution HEAD.tar.gz for the first time

CreatePkg & SignHead

- CreatePkg: Publish HEAD & distribution HEAD.tar.gz for the first time
- SignHead: Sign current HEAD regularly and update package if necessary

CreatePkg & SignHead

- CreatePkg: Publish HEAD & distribution HEAD.tar.gz for the first time
- SignHead: Sign current HEAD regularly and update package if necessary
- Signing should happen regularly (timeouts), even if the HEAD didn't change.

CreatePkg & SignHead

- CreatePkg: Publish HEAD & distribution HEAD.tar.gz for the first time
- SignHead: Sign current HEAD regularly and update package if necessary
- Signing should happen regularly (timeouts), even if the HEAD didn't change.

The administrator has to upload the distribution HEAD.tar.gz to the download URL and publishes the new DNS TXT record in the defined zone. DNSSEC should be enabled.

Install & Update

```
$ dig -t TXT _codechain.codechain.secpkg.net
[...]
_codechain.codechain.secpkg.net. 903 IN TXT "49_tm45BwAdDitsidwlFWXsTOd-PZ71eyskggnXu
ECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABbs-byAAAAAFvbc_IAAAAAAAAAAFPywm2
S4XMwboPVTjED7y4L2HpWExW8S0nh7mx437WDpovEmnX4EpoOHasaPYxTIjIY6x34ygspeLGkvvfVrz6y3U
vfE-QAN61dXzrBjYlx2LNd7xXRGjHopi0Am82kBA"
[...]
```

- Install: install software described by a .secpkg file for the first time.
- Update: Query for updates and perform them, if necessary.

Installing the Codechain package

```
$ secpkg install codechain.secpkg
.secpkg: written
signed head found: 53f2c26d92e173306e83d54e3103ef2e0bd87a561315bc4b49e1ee6c78dfb583
/home/frank/.config/secpkg/pkgs/codechain/signed_head: written
download
http://frankbraun.org/codechain/53f2c26d92e173306e83d54e3103ef2e0bd87a561315bc4b49e1ee6c
env GO111MODULE=on go build -mod vendor -v ./...
env GO111MODULE=on GOBIN=/home/frank/.config/secpkg/local/bin go install \
  -mod vendor -v ./...
```

three Codechain tools for different user roles

1. `codechain` for developers to record code changes and corresponding multi-party reviews in a unmodifiable hash chain.

three Codechain tools for different user roles

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.

three Codechain tools for different user roles

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

Codechain

- Codechain beta is available now

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)
- ≈7000 lines of code (plus vendored dependencies)

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)
- ≈7000 lines of code (plus vendored dependencies)
- public domain (`http://unlicense.org/`)

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)
- ≈7000 lines of code (plus vendored dependencies)
- public domain (`http://unlicense.org/`)
- Codechain depends on the `git` binary (for `git diff`), but that's optional

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)
- ≈7000 lines of code (plus vendored dependencies)
- public domain (`http://unlicense.org/`)
- Codechain depends on the `git` binary (for `git diff`), but that's optional
- Codechain is reviewed and signed with Codechain (2-of-3)

Codechain

- Codechain beta is available now

→ `https://github.com/frankbraun/codechain`

- minimal code base, Go only, cross-platform (tested on Linux)
- ≈7000 lines of code (plus vendored dependencies)
- public domain (`http://unlicense.org/`)
- Codechain depends on the `git` binary (for `git diff`), but that's optional
- Codechain is reviewed and signed with Codechain (2-of-3)

current head of Codechain's hash chain:

`c63c7648696b9017a13b101f917254f9876dcc09d98ee9d1bb83a1e43bdd05b9`

conclusion

While it is good that code signing is widely deployed now it doesn't solve important attack vectors.

conclusion

While it is good that code signing is widely deployed now it doesn't solve important attack vectors.

Codechain mitigates:

- key compromise (with multiparty signatures & key rotation)

conclusion

While it is good that code signing is widely deployed now it doesn't solve important attack vectors.

Codechain mitigates:

- key compromise (with multiparty signatures & key rotation)
- developer coercion (with multiparty signatures & key rotation)

conclusion

While it is good that code signing is widely deployed now it doesn't solve important attack vectors.

Codechain mitigates:

- key compromise (with multiparty signatures & key rotation)
- developer coercion (with multiparty signatures & key rotation)
- mitigates regression / suppression of updates (with SSOT)

conclusion

While it is good that code signing is widely deployed now it doesn't solve important attack vectors.

Codechain mitigates:

- key compromise (with multiparty signatures & key rotation)
- developer coercion (with multiparty signatures & key rotation)
- mitigates regression / suppression of updates (with SSOT)

This gives us

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

binaries build from source, which mitigates targeted updates

acknowledgments: Jonathan "Smuggler" Logan

acknowledgments: Jonathan "Smuggler" Logan

contacts:

- Email: `frank@cryptogroup.net` (use PGP, key on keyserver)
- `94CC ADA6 E814 FFD5 89D0 48D7 35AF 2AC2 CEC0 0E94`
- Twitter: @thefrankbraun

acknowledgments: Jonathan "Smuggler" Logan

contacts:

- Email: `frank@cryptogroup.net` (use PGP, key on keyserver)
- `94CC ADA6 E814 FFD5 89D0 48D7 35AF 2AC2 CEC0 0E94`
- Twitter: @thefrankbraun

acknowledgments: Jonathan "Smuggler" Logan

contacts:

- Email: `frank@cryptogroup.net` (use PGP, key on keyserver)
- `94CC ADA6 E814 FFD5 89D0 48D7 35AF 2AC2 CEC0 0E94`
- Twitter: @thefrankbraun

slides: `http://frankbraun.org/in-code-we-trust-2.pdf`

acknowledgments: Jonathan "Smuggler" Logan

contacts:

- Email: `frank@cryptogroup.net` (use PGP, key on keyserver)
- `94CC ADA6 E814 FFD5 89D0 48D7 35AF 2AC2 CEC0 0E94`
- Twitter: @thefrankbraun

slides: `http://frankbraun.org/in-code-we-trust-2.pdf`

see also essay under `http://frankbraun.org/essay/` (25 pages)

acknowledgments: Jonathan "Smuggler" Logan

contacts:

- Email: `frank@cryptogroup.net` (use PGP, key on keyserver)
- `94CC ADA6 E814 FFD5 89D0 48D7 35AF 2AC2 CEC0 0E94`
- Twitter: @thefrankbraun

slides: `http://frankbraun.org/in-code-we-trust-2.pdf`

see also essay under `http://frankbraun.org/essay/` (25 pages)
thank you very much for your attention! questions?