Securing our Package Distribution System

Duncan Coutts and Edsko de Vries

August 2015, Haskell Implementors Workshop —

Well-Typed
The Haskell Consultants

Whither security?

- Why do we want security at all?
- Why now?
- What do we mean by security?
- What security properties do we actually want?

Security goals

Things we might want

- packages not modified between server and user
- packages not modified between author and user
- packages written by folks we trust

Things to keep in mind

- trade-off between security and convenience
- rather not raise barrier to entry for package authors
- mirrors are useful
- trusting CDNs and mirror operators isn't great

Security system design

Everyone knows we should leave crypto algorithms and protocols to the experts

Security system design is also hard

- Subtle details
- Easy to miss important issues
- Many examples of misunderstandings and poor implementations

Recent example (Jan 2015): docker image “verification”
`https://lwn.net/Articles/628343/`

Security system design

Everyone knows we should leave crypto algorithms and protocols to the experts

Security system design is also hard

- Subtle details
- Easy to miss important issues
- Many examples of misunderstandings and poor implementations

Conclusion

Where possible, leave security system design to the experts

Well-Typed

Security system design

We do use an existing expert design (TUF)

But first, not convinced it’s tricky? Lets try it...

Security system design

We do use an existing expert design (TUF)

But first, not convinced it's tricky? Lets try it...

Obvious approaches:

Authors sign packages

- sign individual tarballs
- authors manage their own keys
- some mechanism for clients to decide which author keys are ok

Server signs a manifest

- manifest lists all tarball names and hashes
- manifest signed by key held on server
- clients trust server key

Server uses HTTPS

Security system design

Obvious advantages and disadvantages:

Authors sign packages

- extra work for authors
- no protection for unsigned packages
- “end to end” – should be resilient to server compromise
- careful design needed on policy for deciding which author keys are ok

Server signs a manifest

- no extra work for authors
- covers all packages
- no protection in case of server compromise

Server uses HTTPS

- cannot use CDN/mirrors
- or, trust CDN/mirrors
- no protection in case of server compromise

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

Potential attacks

Potential attacks from the academic literature

- Modified tarballls
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies an altered version of one of the tarballs
(and gets a client to install it)

Well-Typed

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies an older (but genuine) package, or set of packages (and gets a client to install an older version)

Well-Typed

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

always supplies the same (genuine) set of packages
(preventing the client from discovering that newer versions exist)

Well-Typed

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies combinations of packages (or metadata) that never existed upstream

Well-Typed

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- $\boxed{\text{Extra dependencies}}$
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies altered or additional metadata and gets a client to install extra package dependencies

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies a package by a legitimate author but where that author is not authorised to supply that package

Well-Typed

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

supplies a never-ending stream of data, causing a denial of service

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker

causes the download to be so slow that it is effectively a denial of service

Potential attacks

Potential attacks from the academic literature

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

Imagine the attacker controls the network or mirror/CDN

The attacker can always prevent updates by a denial of service, but it should never go unnoticed.

Well-Typed

How did we do?

Which attacks do our naïve approaches prevent?

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

How did we do?

Which attacks do our naïve approaches prevent?

- $\boxed{\text{Modified tarballs}}$
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- $\boxed{\text{Wrong author}}$
- Endless download
- Slow download

Authors sign packages

- preventing the wrong author attack depends on how we decide which author keys are ok
- cannot simply trust a set of authors

Well-Typed

How did we do?

Which attacks do our naïve approaches prevent?

- $\boxed{\text{Modified tarballs}}$
- Rollback
- Freeze
- $\boxed{\text{Mix and match}}$
- Extra dependencies
- $\boxed{\text{Wrong author}}$
- Endless download
- Slow download

Server signs a manifest

- could prevent extra dependencies if the manifest lists metadata files

Well-Typed

How did we do?

Which attacks do our naïve approaches prevent?

- $\boxed{\text{Modified tarballs}}$
- $\boxed{\text{Rollback}}$
- $\boxed{\text{Freeze}}$
- $\boxed{\text{Mix and match}}$
- $\boxed{\text{Extra dependencies}}$
- $\boxed{\text{Wrong author}}$
- Endless download
- Slow download

Server uses HTTPS

How did we do?

Which attacks do our naïve approaches prevent?

- Modified tarballs
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- Wrong author
- Endless download
- Slow download

The download DOS attacks are a bit different and require specific prevention measures

The Update Framework (TUF)

The Update Framework

The Update Framework (TUF)

- an architecture for secure software update systems
- designed by experts (academics and Tor project members)
- somewhat adaptable for different use cases and to fit existing systems

Overview

- server manifest signing
- plus author signing
- plus extra bits (for replay, freeze and download attacks)

A single coherent and somewhat modular design

Papers

Several papers on the background to TUF

Survivable Key Compromise in Software Update Systems,
J. Samuel, N. Mathewson, J. Cappos, R. Dingledine, CCS 2010

A Look in the Mirror: Attacks on Package Managers,
J. Cappos, J. Samuel, S. Baker, J. Hartman, CCS 2008

Package Management Security,
J. Cappos, J. Samuel, S. Baker, J. Hartman, University of Arizona Tech Report 2008.

Brief details

Roles, keys & metadata files

- root, target, snapshot, timestamp, mirrors
- target role corresponds to author signing
- snapshot & timestamp roles correspond to index signing
- establish a chain of trust back to the root keys

Particular measures to prevent

- rollback attacks
- freeze attacks
- download & DOS attacks

Well-Typed

“Survivable key compromise”

Compromise of a key isn’t necessarily game over

Conversely, different keys provide different security properties (what makes TUF somewhat modular)

Keys have to live in different security contexts to be useful, otherwise compromise of one means compromise of the other.

Different roles have keys that are used in different places and with different frequency, so some are easier to protect than others.

Keys and signatures

TUF is agnostic about

- the cryptographic hash algorithm
- the public key signature algorithm
- but recommends sha256 and ed25519

The formats are extensible to new algorithms over time, and multiple algorithms for smooth transitions.

TUF for Hackage

TUF adaptation for Hackage

Phased implementation

- leaving out one feature for phase 1
- takes advantage of TUF modularity

Integration with existing repository index format

Snapshot and timestamp keys kept on the same server

- impractical with current architecture to keep them separate
- roles not fully merged, leaves open the option to separate the keys later

Phased implementation

Phase 1

- everything but target keys (i.e. no author signing)
- tarballs are protected by the snapshot key
- fully automatic: no extra work for authors or users
- vulnerable if the server is compromised
- funded by the IHG
- in beta now

Phase 2

- delegated target keys: key per developer
- adds partial protection if the server is compromised
- adds some extra work for authors
- opt-in: both signed and unsigned packages
- design already mostly done (so it's compatible)

Seeking funding for phase 2

Integration with Hackage repository format

Existing repository format has a package index

Index contains

- all `.cabal` files by value
- all `.tar.gz` files by reference

Changes

- Hackage index extended with unsigned target metadata files listing `.tar.gz` files' size and hash
- Index file listed in `snapshot.json`

Thus the `.cabal` files and target metadata files are effectively signed by the snapshot key

Well-Typed

Implementation

`hackage-security` library

New `hackage-security` library for use by clients

- implements the update procedure
- not a fully generic TUF implementation
- adapted for Hackage format repositories
- automatic use of mirrors

Three main APIs

- bootstrap client trust using known root key ids
- sync repository info from server (or mirror)
- download an individual package tarball

Also has functionality for servers and other tools

Well-Typed

Library implementation

Structured for clarity, correctness and extensibility

- Layers
  - core TUF types and algorithms
  - repository format
  - HTTP client implementation (parameterised)
- Lots of types, e.g. to track trusted information
- Easy to add new hash algorithms and key types

Well-Typed

Library implementation

Choice of hash and keys

- initially supports sha256 hashes and ed25519 keys
- uses the `ed25519` package (which binds a bundled C implementation by Dan J. Bernstein)
- uses `cryptohash` package (also binds a bundled C impl)

No external dependencies: minimises scope for environment and configuration errors

Verification is always-on, so important that it works every time

Implementation in `cabal-install`

Quite a small patch overall

Uses `hackage-security` library

- for updating the package index
- for downloading package tarballs
- provides own HTTP client implementation, based on `HTTP` package

If necessary, will bundle `hackage-security`, `ed25519` and a SHA library to eliminate any bootstraping dependency issues.

Implementation in `hackage-server`

Uses `hackage-security` library for generating and signing the `timestamp.json` and `snapshot.json`

Timestamp and snapshot keys held in memory

Serves `root.json` and `mirror.json` files directly

Extra goodies

Mirroring

TUF supports mirrors! Our implementation supports mirrors!

How it works

- server supplies `mirrors.json`
- client reads this
- on subsequent updates the client can pick any mirror
- zero configuration required for the client

Client currently has no clever logic to pick mirrors. Should add something smarter if we get regional mirrors.

Host your own repositories

Mirror the public packages

- new `hackage-mirror` tool

Host private repositories

- new `hackage-repo-tool`
- commands to manage & update the repository

Both tools

- produces a local set of files
- can use any standard HTTP server
- use the `hackage-security` library for all the heavy lifting

Incremental updates

Hackage index is now big (10Mb compressed)

`cabal update` times are too long

Extra security metadata makes the index even bigger!

New incremental index update

- repository index updated in append-only manner
- only tail of the index needs to be downloaded
- uses HTTP range requests

Well-Typed

Log based Hackage index

Repository index updated in append-only manner

- can recover earlier states of the repository
- often-requested feature by authors of other tools
- prototype `cabal install --index-wayback=$DATE`

Current Status

Current Status

Alpha release in early July

- github-only, for eager testers

Beta release earlier this week (late August)

- now easy to try out
- deployed on the central `hackage.haskell.org`
- libs released on hackage
- live mirror available

```
cabal install \
  http://www.well-typed.com/hackage-security/Cabal-1.23.0.0.tar.gz \
  http://www.well-typed.com/hackage-security/cabal-secure-beta.tar.gz
```

Well-Typed

Remaining issues

Remaining issues

- a few known issues
- anything arising from the beta test
- details of key management

Well-Typed

Key management

Root keys

- set of root keys
- can issue as many as we like
- we decide the threshold number of keys to re-sign root info
- clients need to be shipped with root key ids
- need to be able to bootstrap in N years time

Basic plan

- Haskell.org committee hold keys (and/or delegates)
- issue 10 keys
- threshold of 3 keys to re-sign root info
- issue operational keys for 6-12 months
- require root keys be held offline?

Details of procedures to be finalised

Credits

Industrial HASKELL GROUP

OSL OPEN SOURCE LAB

Work funded by the IHG members

OSL are providing a public mirror

Edsko de Vries wrote almost all the code

Feedback from alpha testers (particularly Herbert Valerio Riedel)

Thanks!

Questions?

Extra slides: TUF details

How it works

Role	Key(s)	Metadata file(s)
Root	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
Snapshot	single snapshot key	`snapshot.json`
Timestamp	single timestamp key	`timestamp.json`
Mirrors	single mirrors key	`mirrors.json`

How it works

Role	Key(s)	Metadata file(s)
$\boxed{\text{Root}}$	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
Snapshot	single snapshot key	`snapshot.json`
Timestamp	single timestamp key	`timestamp.json`
Mirrors	single mirrors key	`mirrors.json`

- root role delegates trust for the other roles
- `root.json` file lists all the keys
- root keys sign the `root.json` file

Forms the root of trust in the system. Clients need to know (and trust) the root keys.

How it works

Role	Key(s)	Metadata file(s)
Root	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
Snapshot	single snapshot key	`snapshot.json`
Timestamp	single timestamp key	`timestamp.json`
Mirrors	single mirrors key	`mirrors.json`

- target role secures individual “target” files (e.g. tarballs)
- `targets.json` file lists all target files, names, sizes and hashes
- target key signs the `targets.json` file

Well-Typed

How it works

Role	Key(s)	Metadata file(s)
Root	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
$\boxed{\text{Snapshot}}$	single snapshot key	`snapshot.json`
Timestamp	single timestamp key	`timestamp.json`
Mirrors	single mirrors key	`mirrors.json`

- snapshot role secures all metadata in the repository
- `snapshot.json` file lists all metadata files (except the timestamp), names, sizes and hashes
- snapshot key signs the `snapshot.json` file

TUF allows extra custom metadata files

Well-Typed

How it works

Role	Key(s)	Metadata file(s)
Root	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
Snapshot	single snapshot key	`snapshot.json`
$\boxed{\text{Timestamp}}$	single timestamp key	`timestamp.json`
Mirrors	single mirrors key	`mirrors.json`

- timestamp role ensures the freshness of metadata
- `timestamp.json` file lists the `snapshot.json` size and hash
- timestamp key signs the `timestamp.json` file with an expiry time in the near future

The short validity period ensures limited freshness.

Well-Typed

How it works

Role	Key(s)	Metadata file(s)
Root	set of root keys	`root.json`
Target	target key & delegated keys	`targets.json`
Snapshot	single snapshot key	`snapshot.json`
Timestamp	single timestamp key	`timestamp.json`
$\boxed{\text{Mirrors}}$	single mirrors key	`mirrors.json`

- mirrors role is for the secure distribution of a list of mirrors
- `mirrors.json` file lists the repository mirrors
- mirrors key signs the `mirrors.json` file

The mirrors role is optional and not security critical as TUF does not place any trust in mirrors.

Well-Typed

The update process

The client

- reads local `root.json` to find expected keys etc
  - or must bootstrap using known root key ids
- downloads and verifies `timestamp.json`
  - `timestamp.json` refers to `snapshot.json`
- downloads and verifies `snapshot.json` (if it changed)
  - `snapshot.json` refers to `00-index.tar.gz` and other metadata files
- downloads and verifies `00-index.tar.gz` (if it changed)
- now has all package metadata (.cabal files etc)
- now knows expected hashes of all tarballs

Downloading and verifying tarballs is now straightforward

Well-Typed

Download DOS detection

Endless download attack prevention

- except for `timestamp.json`, we know the size of a file before we download it
- `timestamp.json` has bounded size
- must fail during download if we get more data than expected

Slow download attack prevention

- must place a lower limit on download speeds

“Survivable key compromise”

Compromise of

- timestamp key: attacker can do freeze attacks
- timestamp + snapshot key:
  attacker can do freeze, rollback, mix and match attacks
- timestamp + snapshot + mirrors key:
  attacker can supply a bogus list of mirrors
- timestamp + snapshot + target key:
  attacker can change tarballs
- a threshold of root keys:
  game over, attacker can issue new keys for all roles

Well-Typed

Key use and storage

Key	Use frequency	Location
Root	infrequently, manual	may be offline
Target (single)	frequently, automatic	online
Snapshot	frequently, automatic	online
Timestamp	frequently, automatic	public-facing machine
Mirrors	infrequently	offline
Target (primary)	infrequently, manual	offline
Target (delegated)	infrequently, manual	on owners' machines

Well-Typed

Metadata formats

All TUF metadata is in “Canonical JSON” format

- Subset of JSON (e.g. no floats, limited number ranges)
- Canonical form ignores whitespace, sorts keys etc
- Consistent content hashes allows inline signatures

Signed files are of the form

```
{ signatures: [ { keyid:  "4e4e5ae824c86bfd8fb...,
                  method: "ed25519",
                  sig:    "xFVwbpGjjxb2Wv4Dj+s...
                }
              ],
  signed: {...}
}
```

Well-Typed