Securing our Package Distribution System

Duncan Coutts and Edsko de Vries

August 2015, Haskell Implementors Workshop — Copyright © 2015 Well-Typed LLP
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
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 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 altered version of one of the tarballs
(and gets a client to install it)
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)
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)
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
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 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
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.
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?

- **Modified tarballs**
- Rollback
- Freeze
- Mix and match
- Extra dependencies
- **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
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

Server signs a manifest

- could prevent extra dependencies if the manifest lists metadata files
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

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 (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
Several papers on the background to TUF

*Survivable Key Compromise in Software Update Systems*,

*A Look in the Mirror: Attacks on Package Managers*,

*Package Management Security*,
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
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.
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
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
Implementation
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
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
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 bootstrapping dependency issues.
Implementation in hackage-server

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

Timestamp and snapshot keys held in memory.

Serves `root.json` and `mirror.json` files directly.
Extra goodies
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
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
Remaining issues

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

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

<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td>Target</td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td>Timestamp</td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>
## How it works

<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Root</strong></td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td>Target</td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td>Timestamp</td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td><strong>Target</strong></td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td>Timestamp</td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>

- 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
How it works

<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td>Target</td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td>Timestamp</td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>

- 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
### How it works

<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td>Target</td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td><strong>Timestamp</strong></td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>

- 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.
<table>
<thead>
<tr>
<th>Role</th>
<th>Key(s)</th>
<th>Metadata file(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>set of root keys</td>
<td>root.json</td>
</tr>
<tr>
<td>Target</td>
<td>target key &amp; delegated keys</td>
<td>targets.json</td>
</tr>
<tr>
<td>Snapshot</td>
<td>single snapshot key</td>
<td>snapshot.json</td>
</tr>
<tr>
<td>Timestamp</td>
<td>single timestamp key</td>
<td>timestamp.json</td>
</tr>
<tr>
<td>Mirrors</td>
<td>single mirrors key</td>
<td>mirrors.json</td>
</tr>
</tbody>
</table>

- 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.
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.
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
## Key use and storage

<table>
<thead>
<tr>
<th>Key</th>
<th>Use frequency</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>infrequently, manual</td>
<td>may be offline</td>
</tr>
<tr>
<td>Target (single)</td>
<td>frequently, automatic</td>
<td>online</td>
</tr>
<tr>
<td>Snapshot</td>
<td>frequently, automatic</td>
<td>online</td>
</tr>
<tr>
<td>Timestamp</td>
<td>frequently, automatic</td>
<td>public-facing machine</td>
</tr>
<tr>
<td>Mirrors</td>
<td>infrequently</td>
<td>offline</td>
</tr>
<tr>
<td>Target (primary)</td>
<td>infrequently, manual</td>
<td>offline</td>
</tr>
<tr>
<td>Target (delegated)</td>
<td>infrequently, manual</td>
<td>on owners’ machines</td>
</tr>
</tbody>
</table>
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

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