Conex - establishing trust into data repositories

Hannes Mehnert
University of Cambridge

Louis Gesbert
OCamlPro

Abstract

Opam is a software update systems, responsible for discovering, downloading, building, and installing packages. The data and metadata at each step should be authenticated to originate from a (transitively) trusted author. Opam does not include any mechanism to authenticate any step at the moment. We propose *conex*, which establishes end-to-end digital signatures from the package author to the user. Using conex, neither the distribution server nor the transport layer need to be trusted. Authors and users need to adapt their workflow only slightly (or use opam-publish), whereas janitors (repository maintainers) need more tooling.

1. Introduction

Opam discovers, downloads, builds, and installs OCaml packages. To prevent injection of malware, all received data needs to be authenticated. A software update system relies on a central server which distributes a repository (package metadata: dependencies and build instructions) of packages. When a new package is released, it is inserted into the repository and distributed to all clients by the central server. The central server is an online service and a single point of failure. Recently those servers have been under attack [2–7, 10–23, 26–28, 31] to inject malware into packages.

So far there has been no breach of opam detected, but other package managers have been compromised, see [7, 15, 17, 27]. Opam does not provide any security mechanism at any step: Discovery in current opam (1.2.2) is done via a https download (for the main opam repository), but the download tool (`wget`) is used with the `--no-check-certificates` command line option, disabling any certificate validation. Thus, a person-in-the-middle can inject arbitrary metadata (additional patches, modified packages, modified checksums, ...) during the discovery step. A package is downloaded using the URL specified in the metadata of the package (usually https, again without certificate validation), and checks the recorded MD5 digest (which is known to be weak [25]) against the downloaded archive. The build step is not contained into a temporary chroot environment, but may modify arbitrary files on disk. The recipe for building is part of the metadata, and may also include invoking any shell commands (such as downloading more things). The install step may also modify arbitrary files.

Opam 2.0 is not released yet, but fixes major concerns: downloading validates the TLS certificates [24] using system-wide trust anchors, a wrap command is available for building and installing packages [1], optionally enabling sandboxing. Nevertheless, a breach of the central server enables an attacker to modify packages arbitrarily. Relying only on TLS for packages is suboptimal, because many TLS implementations suffer from a long history of security vulnerabilities (e.g. goto fail).

In this paper we present *conex*, a system which uses end-to-end digital signatures - the original author signs their package and build instructions, the client verifies this signature - to re-establish trust into packages installed by opam. Using end-to-end signatures, neither the the server hosting the repository nor the server where the tarballs are downloaded from need to be trusted! Conex has been designed with opam in mind, but it can be used without opam to publish and update any data in an authenticated way. Conex is not limited to the main opam repository, but can be used with any public or private opam repository, or any other data repository. The only requirement is a second channel to distribute an initial set of trust anchors (e.g. an opam package in the main opam repository). The main goal of conex is to *establish an authenticated way to distribute packages*. The main opam repository is a git repository hosted on GitHub. Conex does not depend on GitHub, or even on git. Conex is constrained by the current use of the main opam repository: most package authors release a package and submit a pull request (PR). Janitors (repository maintainers) then check the PR, TravisCI builds it using several OCaml and opam versions, and the automated bot Camelus [8] reports opam inconsistencies.

Conex is implemented in OCaml and licensed under the 2-clause BSD license (source [9]). To provide authenticity from the beginning, conex is best installed together with opam.

Earlier work on securing software update systems, such as the update framework (TUF) [35] and Diplomat [33] focus on distributing directories securely, but due to centralised files simultaneous updates of packages are not free of conflicts. Our earlier proposal for opam signing [29] introduced unnecessary structure and complexity by adjusting TUF to the decentralised opam workflow.

In section 2 we discuss the design of conex, followed by reasonable system optimisations in section 3. Afterwards in section 4 we briefly present the implementation of conex and opam integration. Related work is mentioned in section 5, we conclude in section 6.

2. Design

Conex does not claim to protect against undiscovered vulnerabilities in the packages, dependent tools, or conex itself. It is also not a scanner for installed vulnerable packages.

2.1 Threat model

There are many risks that users of software update systems face, ranging from injecting traffic over weaknesses in TLS, weaknesses in the network infrastructure, compromising signing keys, to compromising the repository or signing infrastructure.

This leads us to consider a threat model where some involved systems are compromised. We assume that an attacker can:

- Compromise the central server distributing the repository.
- Respond to user requests (acting as person-in-the-middle).

An attack is successful if the attacker can change the contents of a package that a user installs, or preventing a user from updating to the most recent version of a package.

We cannot detect compromises of (offline!) keys, but this requires janitors, authors, and other people to carefully read through updates. We assume that authors and janitors protect their private keys on their computers in a reasonable way, but we also have mechanisms in place for key revocation. If a private key is compromised, we consider conex security to be effective if the impact is limited to the authorised packages of the compromised key.

2.2 Roles

There are two roles in conex:

- An *author* who develops packages and releases them to the repository. Each author signs their own packages.
- A *janitor* who keeps the repository tidy and in a working state (fix up reverse dependencies etc.). Janitors also maintain the package name and author name resources by signing them. Authorisations, that an author is responsible for a package, are signed by janitors as well. Instead of trusting a single janitor to sign these resources, a quorum of signatures of janitors is required to avoid this concentration of power.

It is likely that a janitor acts as an author for their packages, there is no need for a janitor to have multiple private keys.

2.3 Resources

We distinguish various resources which are all hosted in the same repository: package metadata (checksums, build recipe, dependencies), author information (public key, id, role), list of releases (of a single package), and authorisation (relation between package name and author). The distinction between an author and a janitor is done via the role of the author information. The repository contains one index per janitor with all resources and checksums the janitor vouches for.

Each resource is only valid if it is signed by either one of the authorised key identifiers, or a quorum of janitors. Each resource contains a monotonic counter to prevent rollback attacks.

There are two universes of names which need to be protected against starvation attacks (and typographic ones [30]): package names and key identifiers. Janitors need to carefully check new package and public key submissions.

Each author updates their own packages, which should be free of conflicts between concurrent repository updates, due to separate resources for each package. If a conflict is detected, it is up to the janitors to decide which update should go through, and which should be resubmitted on top of the other.

2.4 Repository layout

Conex is not bound to the file system, but we use their terminology. The root directory has three subdirectories, `keys`, `sigs` and `data`.

The `keys` directory contains one file for each author and janitor, named after the key identifier. The `sigs` directory contains one file for each janitor, named after the janitor key identifier. The `data` directory consists of subdirectories, which each must contain a file `authorisation` and a file `releases`, plus possibly any subdirectories. Each subdirectory must be listed in the `releases` file, and must contain a `checksum` file besides any other data files and directories (all listed in the checksum file).

Each file contains a set of key-value pairs. Any key-value representation which can be non-ambiguously normalised into a string, can be used (such as the opam format, json, s-expressions). Each file contains a `signed` structure: `(signed:` **resource**`, sigs:` **(id, signature) list**`)`. While signatures from authorised authors are done directly on the checksum/release/public key files, each janitor signs only their per-janitor index file. This reduces the amount of signature verifications which are done by the client. The different resources are explained in more detail below. The signature is computed over the normalised string representation of the resource concatenated with the public key id and the resource kind, separated by a space character.

id: restricted string
name: restricted string
signature: Base64 encoded RSASSA-PSS with SHA256 (PKCS1)
public key: RSA public key ($\geq$ 2048 bit, PKCS1, PEM encoded)
role: `Janitor` | `Author` | `Other` of **id**
service: `Email` of **name** | `GitHub` of **name** | `Other` of **id** * **name**
digest: SHA256 digest of file
byte size: size as **Int64** of file

Figure 1. Data types for conex version 0

The data types for version 0 of conex are presented in Figure 1. A `restricted string` is a case insensitive 7 bit ASCII string without control or white space characters. Depending on usage, more characters are forbidden (e.g. opam package names may not contain a '.'). Distinction of id and name is only for clarity (ids are the public key ids, where names are the package names), they form two disjoint sets of identifiers. Upon insertion of new identifiers, checks for non-collision and validity need to be done.

Each file includes a data version (0 in this proposal) and a monotonic 64 bit counter (starting at 0).

counter: **Int64**,
version: **Int64**,
key: **public key**,
identifier: **id**,
accounts: **service list**,
role: **role**

Figure 2. Public key

counter: **Int64**,
version: **Int64**,
name: **name**,
authorised: **id list**

Figure 3. Authorisation

The file format for a public key, presented in Figure 2, contains the PEM encoded public key, the identifier (used for signatures and authorisations in opam), accounts in other systems, and its role.

The authorisations file format, shown in Figure 3, contains the package name and a list of authorised key identifiers for this package.

counter: **Int64**,
version: **Int64**,
name: **name**,
releases: **name list**

Figure 4. Releases

counter: **Int64**,
version: **Int64**,
name: **name**,
files: **(name, byte size, digest) list**

Figure 5. Checksum

The releases file format (Figure 4) contains the package name and a list of all releases. It is signed by an authorised author or a quorum. This is crucial to avoid rollback attacks, where an attacker hinders a client from receiving package updates.

The checksums file format, shown in Figure 5, contains the release name, and a list of file names which are part of the release (opam, optionally patches, archive, ...), their sizes and digests. This list must contain all files recursively present in the current directory (excluding the checksum file).

counter: **Int64**,
version: **Int64**,
identifier: **id**,
resources: **(name, kind, digest) list**

Figure 6. Janitor index

The janitor index file format, shown in Figure 6, contains the janitor identifier, and all resources the janitor vouches for. It may contain any checksums, even of resources not yet in the repository (but proposed in some pull request). This avoids merge conflicts if a janitor wants to vouch for multiple new authors and packages at the same time.

2.5 Chain of Trust

A package is valid if and only if the chain of trust can be verified. All files in the subdirectory are registered in the checksum file, which is signed by either an author (listed in the authorisation file in the parent directory) or a quorum of janitors. The subdirectory name must be present in the valid releases file of the parent directory. The authorisation file itself has to be signed by a quorum of janitors. Each public key has to be signed by its private key and a quorum of janitors.

To bootstrap the chain, we assume an existing set of janitors (which cardinality is above the quorum). The set for the main opam repository is distributed with opam. The opam installation, including the initial set of janitors, is out of scope for conex (e.g. by another package manager, OpenPGP signatures).

2.6 Verification

Since these verification steps are done by automated systems (see section 3) as well, verification errors are not expected at the client and treated as fatal. The user is informed about the potential security breach, the set of patches are not applied locally, and the user is asked to inform the janitors via email. Certainly, the user can try again to update their repository.

A client can verify a repository in two ways:

- a *complete snapshot* of the repository (fresh clone)
- an already verified repository and a list of *updates* which should be applied

We will briefly outline the algorithm for both subsequently.

Snapshot verification Verification of a snapshot is straightforward: take the set of initial janitor keys as trusted, verify all janitor index files, iterate over all packages, verify the authorisation file (properly signed by a quorum of janitors), verify all releases files, and verify all checksum files. The client also needs to ensure that there are no subdirectories or files which are not listed in the releases or checksum files. The public keys are verified upon demand: whenever a signature is encountered, the key of the signing id is verified.

Update verification The repository is already verified. The client uses it to verify each update separately. Each modified resource must be signed, either by an authorised author or by a quorum of janitors. The counter must start at 0 (upon creation) and increment. The file name and identifier must match. A key rollover needs to introduce the new key (replacing the old), and sign again all resources which were signed with the old key. Removal of resources is done by replacing it with the empty string. A new package needs to have an authorisation file and a releases file, both properly signed.

2.7 Security Analysis

If package metadata and public keys are distributed from different sources, an attacker who compromised a key could hinder a user from receiving the key revocation by preventing communication to the key server. This would allow the attacker to continue signing packages with the compromised key and having the user install arbitrary data. To avoid this, package metadata, signatures, and public keys are in the same repository.

Each resource includes a monotonically increasing counter to prevent rollback attacks. Otherwise, an attacker could replay e.g. an insecure package build recipe. Also, each resource includes a name, otherwise the signature of package A would be valid for package B. Each signature contains the resource kind to avoid carrying over a signature from one resource to another.

Instead of removing resources to revoke them, the already claimed names stay there. The public key, package, or janitor index resource is emptied and must be signed by a quorum of janitors.

A person-in-the-middle attacker could implement a backdoor for a package, but would need to either modify the checksum, prepare a new release, embed the patch as metadata in the repository, or introduce a new package (which they could) and add a dependency. All of these actions require either a signature from the original author or from a quorum of janitors. Any suspicious behaviour in widely used packages will likely be spotted by the community which reads through the commit logs.

An attacker can freeze a client from updating (by preventing the client to communicate with the repository server), but downgrades of packages are not possible (an update cannot remove packages without appropriate signatures).

Conex does not depend on unreliable system time and comparison of timestamps, instead it uses monotonic counters. Other systems rely on git signing, which is out-of-tree data and thus an attacker can prevent a user from receiving this data (while still receiving (potentially malicious) updates).

In the case conex cannot verify package metadata, it will not pass this unverified data to opam. In case conex cannot verify a downloaded tarball, this will be reported.

Mix-and-match attacks provide e.g. outdated releases of package A and B, up-to-date releases of package C, are not mitigated in conex since there is no central list of releases. They can be mitigated by relying on some append-only log, blockchain, or other monotonic updates, such as git without forced pushes. We will discuss a timestamp notary in the next section, mitigating both mix-and-match and freeze attacks.

3. Optimisations

To mitigate mix-and-match attacks, a *timestamp notary*, which updates the repository regularly, verifies all updates, and attaches its digital signature to the latest patch (via an git annotated tag over the git commit). The public key of the timestamp notary is inside the repository and signed by multiple janitors. Clients can, instead of verifying packages and updates independently, trust the timestamp notary (or multiple timestamp notaries). This will reduce computational overhead (signature verification are expensive operations on big numbers), and might be worth for embedded devices. If there are multiple instances of a timestamp notary run by different organisations, and its code is well audited, it might be reasonable to use them as default. While there is not yet an implementation of a timestamp notary, it will likely be implemented as a MirageOS [32, 34] application, thus the trusted code base is small.

The main opam repository served as an example over the last section, but conex can easily be applied to any other opam repository. Initial set of janitors (can be a single element), the quorum (can be one) must be distributed for each other repository. Of course, an automated timestamp notary can be used for other repositories as well.

If the work overhead for janitors turns out too high, we can leverage the system in a way that *neither authorisation nor public keys require a quorum*. This will open the system to name squatting attacks. Snapshot verification does not apply anymore, because there is no history of authorisations. Update verification can still be done, thus we could trust the timestamp notaries and use git.

Another useful automated system will be a *PR bot*, similar to Camelus, which verifies each PR on the main opam repository and reports the results of the update (whether it is valid, how many janitors need to sign, etc.).

Automated systems which build a package and record the (platform-dependent) digests of the resulting binaries can be integrated to sign their results in the repository. Since OCaml 4.03 produces reproducible binaries, thus the locally produced binaries can be checked. Conex can also serve for binary distribution.

4. Conex implementation

Conex is work-in-progress [9]. It will be functorised over the data layout (json, opam, s-expression), and over a cryptography provider. One provider will be nocrypto, another the *openssl* utility, enabling bootstrapping without opam. The timestamp notary and other automated tools will use the conex library.

Conex is not yet integrated into opam, but work on opam 2.0, such as compilers-as-packages, are prerequisites for conex. The opam publish utility will support authors: creating private keys, creating checksum, authorisation, and releases files, and signing these. There is a demand for tooling for janitors, such as a queue of updates requiring signatures for a quorum.

5. Related work

Conex is based on the update framework [35], which centralises all packages into a single `target.txt` file. Instead of doing this, conex was designed with distributed updates in mind.

More recent is Diplomat [33], which automatically promoting unclaimed packages to claimed (which we can do using timestamp notary). Our security model instead uses end-to-end signatures of packages, and relies on authors to sign their packages.

The Haskell signing proposal does not enforce authors to sign their packages, but the repository is signed with an online key.

Our original opam signing proposal [29] did not include a list of releases per package, opening up to various attacks. Additionally, our original proposal introduced an unnecessary hierarchical structure. It relied on the timestamping notary or using git as update protocol with appropriate protection of the transport layer.

6. Conclusion

We presented conex, which secures the software update system opam with minor modifications to the current workflow. Conex uses signatures from package authors to users, removing the demand to trust a central server. Conex mitigates known practical attacks as well as theoretical ones described in literature.

Future work includes finishing the implementation of conex, integrating it into opam, and developing automated services (timestamp notary, build bot, PR bot). A first beta version of conex, including opam integration, is expected to be available for the OCaml workshop 2016.

Acknowledgements Parts of this work were supported by EPSRC Programme Grant EP/K008528/1 (REMS - Rigorous Engineering of Mainstream Systems). We thank especially Joe and Raphaël for their feedback on earlier drafts.

References

[1] Add global 'wrap-build/install/remove-commands:' fields https://github.com/ocaml/opam/pull/2563.

[2] Adobe to revoke code signing certificate https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html.

[3] apache.org incident report for 04/09/2010 https://blogs.apache.org/infra/entry/apache_org_04_09_2010.

[4] apache.org incident report for 08/28/2009 https://blogs.apache.org/infra/entry/apache_org_downtime_report.

[5] Attackers sign malware using crypto certificate stolen from opera software http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/.

[6] An attempt to backdoor the kernel http://lwn.net/Articles/57135/.

[7] Cabal: Packages are downloaded insecurely https://github.com/haskell/cabal/issues/936.

[8] Camelus: posting opam analysis reports on opam-repository pull-request https://github.com/AltGr/Camelus/.

[9] Conex git repository https://github.com/hannesm/conex.

[10] The cracking of kernel.org http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg.

[11] Debian investigation report after server compromises https://www.debian.org/News/2003/20031202.

[12] Freebsd.org intrusion announced 17-11-2012 http://www.freebsd.org/news/2012-compromise.html.

[13] A further update on php.net http://php.net/archive/2013.php#id2013-10-24-2.

[14] Gnu savannah compromise https://savannah.gnu.org/maintenance/Compromise2010/.

[15] How to take over the computer of any java (or clojure or scala) developer http://blog.ontoillogical.com/blog/2014/07/28/how-to-take-over-any-java-developer/.

[16] Important: Information regarding savannah restoration for all users https://savannah.gnu.org/forum/forum.php?forumid=2752.

[17] Newly paranoid maintainers http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers.

[18] The php project and code review http://bjori.blogspot.com/2010/12/php-project-and-code-review.html.

[19] phpmyadmin corrupted copy on korean mirror server https://sourceforge.net/blog/phpmyadmin-back-door/.

[20] php.net security notice http://www.php.net/archive/2011.php#id2011-03-19-1.

[21] Public key security vulnerability and mitigation https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation.

[22] Recent reboots explained: the forge has been compromised https://forge.ocamlcore.org/forum/forum.php?forum_id=913.

[23] Red hat security advisory: openssh security update https://rhn.redhat.com/errata/RHSA-2008-0855.html.

[24] remove insecure / no-check-certificate flags https://github.com/ocaml/opam/pull/2460.

[25] Rfc 6151: Updated security considerations for the md5 message-digest and the hmac-md5 algorithms https://tools.ietf.org/html/rfc6151.

[26] rsync.gentoo.org: rotation server compromised https://security.gentoo.org/glsa/200312-01.

[27] Rubygems data verification http://blog.rubygems.org/2013/01/31/data-verification.html.

[28] Security incident on fedora infrastructure on 23-01-2011 https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html.

[29] Signing the opam repository https://opam.ocaml.org/blog/Signing-the-opam-repository/.

[30] Typosquatting programming language package managers http://incolumitas.com/2016/06/08/typosquatting-package-managers/.

[31] Wordpress passwords reset https://wordpress.org/news/2011/06/passwords-reset/.

[32] Kaloper-Meršinjak, D., Mehnert, H., Madhavapeddy, A., and Sewell, P. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 223–238.

[33] Kuppusamy, T. K., Torres-Arias, S., Diaz, V., and Cappos, J. Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 567–581.

[34] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013), ACM.

[35] SAMUEL, J., MATHEWSON, N., CAPPOS, J., AND DINGLEDINE, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 61–72.