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 to use more sophisticated tools.

1. Introduction

Opam is a software update systems, and needs to discover, download, build, and install packages. To prevent injection of malware, all steps need to authenticate the received data before processing it. A software update system relies on a central server which distributes a repository (source code and metadata such as dependencies and build recipe) of all 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, and recently these central servers have been attacked [2–7, 10–23, 26–28, 31] to distribute packages with injected malware.

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 several major concerns: downloading now validates the TLS certificates [24] using the system-wide trust anchors, a wrap command is available for building and installing packages [1]. Nevertheless, a breach of the central server enables an attacker to modify packages arbitrarily. Relying only on transport layer security for packages, where many implementations have a long history of security vulnerabilities (e.g. goto fail), is not optimal.

* Funded by EPSRC Programme Grant EP/K008528/1

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 data in an authenticated way. At the moment, the main opam repository is a git repository hosted on GitHub. Conex, as presented here, uses GitHub identifiers, but it can use email addresses instead. Conex does not depend on GitHub, neither on git. 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*. Conex is constrained by the current use of the opam repository: most package authors release a package and submit a pull request (PR) to the main opam repository on GitHub. Afterwards, janitors (repository maintainers) check the PR (good package name, potentially missing dependencies or build-time only dependencies), automated builds are performed by TravisCI, and the automated bot Camelus [8] reports feedback and missing fields. Using conex, most of the burden has to be carried by the janitors.

Conex is implemented in OCaml and licensed under the 2-clause BSD license (source [9]). To provide authenticity from the start, conex should be installed at the same time as opam, thus it is constrained to use few dependencies, and will only depend on cmdliner and opam-lib.

Earlier work on securing software update systems, such as the update framework (TUF) [35] and Diplomat [33] focus on distributing directories securely, but make simultaneous updates of several packages by different authors hard, due to centralised files with authorisation and key information. Our earlier proposal for opam signing [29] adjusted ideas from TUF to the current opam workflow, but introduced superfluous structure.

In section 2 we will discuss the design of conex, followed by reasonable shortcuts in section 3. Afterwards in section 4 we present the implementation and opam integration. We will discuss related work in section 5 and conclude with future work in section 6.

2. Design

We first take a step back and do not consider all the details of opam for now.

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 interjecting traffic over weaknesses in TLS, weaknesses in the network infrastructure, compromising signing keys, stealing keys, to compromising the repository or signing infrastructure.

This leads us to consider a threat model where parts of the 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.
- 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.

An author signs their own packages. A janitor introduces authors, authorise authors for their packages, and also adjust packages due to updated dependencies (such as the version constraints, or minor patches to build on another platform). 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. In section 3 we discuss ways to make conex easier to use, leveraging the security level.

If a single janitor would be trusted to tidy up packages, compromising only one janitor key would be sufficient to undermine the security of conex. We avoid this by requiring a quorum of janitors to sign hotfixes.

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. For each janitor the repository contains a resource with all resources and checksums, signed by the janitor (to avoid way too many public key operations during sign and verify).

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 we need to protect against squatting attacks (including typographic ones [30]): package names and key identifiers. Janitors need to faithfully verify new package and public key submissions.

Each author updates their own packages, which should be free of conflicts 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.

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

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

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

2.4 Repository layout

While conex is not bound to a file system, we'll reuse their terminology. The root directory has three subdirectories, `keys`, `sigs` and `data`.

The `keys` directory contains one file for each author, 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**`)`. 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.

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 characters or special 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).

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

Figure 6. Janitor index

The releases file format, shown in Figure 4, contains the package name and the list of all releases of this package. It is signed by an authorised author (or a quorum of janitors). 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, url, descr, optionally patches, archive) and their sizes and digests. This list must contain all files recursively present in the current directory (excluding the checksum file itself).

The janitor index file format, shown in Figure 6, contains the janitor identifier, and all resources the janitor vouches for.

Instead of removing author ids and packages, we put empty placeholders signed by janitors into the repository.

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. Public keys have to be signed by a quorum of janitors as well.

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 initial authenticated opam installation, including the initial set of janitors, is out of scope for conex (requires another package manager, OpenPGP signatures, ...).

2.6 Verification

Both a snapshot of the repository can be verified using a set of initial trust anchors (public keys of janitors), and an update (patch) to an existing verified repository which leads to a verified repository with the patch applied.

Snapshot verification Verification of a snapshot is straightforward: take the set of initial janitors as trusted, then iterate through all packages, verify the authorisation file (properly signed by a quorum of janitors), all releases files, and verify all checksum files. The public keys are verified upon demand: whenever a signature is encountered, the key of the signing id is verified.

Update verification Consider that the repository in the old state is verified, use it to verify the update (in form of a patch) in the following way:

- key modification – key is sufficiently big, unique id matching filename, counter is increased, signed by quorum, no signature with old key
- key insertion – key is sufficiently big, unique id matching filename, counter is 0, signed by quorum
- key deletion – counter increasing, no signatures with this key, empty string is used to denote no key, signed by quorum
- role modification – counter increasing, no signature with old role, signed by quorum
- authorisation insertion – counter is 0, id matches directory, signed by quorum, authorised key ids are present
- authorisation modification – counter increasing, all checksum are signed with an authorised key, signed by quorum
- complete package removal – empty authorised, empty releases, signed by quorum
- package insertion – valid authorisation, releases, and checksum, both counter 0, signed by author (authorisation by quorum)
- package update – valid checksum, counter increasing, signed by author
- new package version release – index counter increasing and new version added to list, valid checksum, signed by author
- package version deletion – counter increasing, removed from releases, signed by author

2.7 Security Analysis

Package metadata, signatures, and public keys are distributed in the same repository to prevent attackers from hindering one communication link, e.g. the public key one. This would otherwise allow a detected attacker who compromised a key to deliver metadata signed with this compromised key.

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 the 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. Shortcuts

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 service is inside the repository and signed by multiple janitors. Clients can, instead of verifying packages and updates independently, trust the timestamp service (or multiple timestamp services). This will reduce computational overhead (signature verification are expensive operations on big numbers), and might be worth for embedded devices. If there are multiple timestamp notaries 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 for identifiers. In a snapshot authorisations cannot be verified anymore, because there is no history to the authorisation. Update verification still applies: if we trust the timestamp notaries and use git, janitors don't have to sign public keys and authorisation files.

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 does not depend on opam-lib, but instead will be functorised over the data layout (json, opam, s-expression). To further minimise the dependencies, the goal is to have a cryptographic module which uses the widely deployed *openssl* utility for bootstrapping.

The timestamp notary and other automated tools will use the conex library as a basis.

Conex is not yet integrated into opam, but some work on opam 2.0, such as compilers-as-packages, will make the integration smooth. The idea is to use conex as a separate application which proxies discover and download commands done by opam.

There is still demand for more tooling support for janitors. In order to reach a quorum, individual janitors have to add the digest to their index file. Once the quorum is reached, the proposed PR can be merged. A queue of updates needing attention (a quorum) will be provided by the tooling.

5. Related work

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

A more recent contribution is Diplomat [33], but instead of automatically promoting unclaimed packages to claimed (which we can do using timestamp services), we believe in end-to-end signing of packages. Our security model relies on offline keys which are protected by authors.

The Haskell signing proposal and implementation does not enforce authors to sign their packages, but instead the repository is signed. Thus, the repository server (or key server) needs to be trusted ultimately.

Our original opam signing proposal [29] did not include a list of releases per package, opening up to attacks where the user does not receive a complete list of releases. Additionally, the original proposal introduced an unnecessary hierarchy of keys, and complicated the setup. It relied on the timestamping notary or to use 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 end-to-end signatures (from package authors to users) as trust model. It removes any trust into a central server. Known attacks, observed in practise or described in literaturre, are mitigated sufficiently by conex.

Future work includes finishing the implementation of conex, integrating it into opam, and develop automated services (timestamp notary, build bot, PR bot).

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.