–Independent Work Report Fall, 2015–

Coniks 2.0:

Publicly Verifiable Keystore with

Key Changing and Verifying Capabilities

Michael Rochlin BSE COS '16

April 30, 2015

Abstract

A requirement of public-key infrastructure is that users can verify that a key belongs to a specific person. While it is infeasible for machines to verify a "correct" user-key binding, we assume that simply having continuity of name-key bindings is sufficient for secure communications.

In this paper and corresponding reference implementation we present CONIKS 2.0, an extension of the original CONIKS server and client which only supported registration and verification, but did not allow for key changes. The new server and client repurpose CONIKS to allow saving and verifying the continuity of any data, along with the ability to verifiably change that data. The new system allows CONIKS to be usable in the real-world for a variety of applications.

1. Introduction

People around the world use the Internet for everything from banking to social networking. Instead of individually handling these complex systems, users rely on and place trust in a variety of providers. Much of our Internet use relies on the security and accuracy of systems and we rely on the providers to provide these guarantees. However, recent cyber-attacks and revelations about mass online surveillance has damaged our trust in providers. Users must either choose to forgo the services of providers or be exposed to the risks associated with trusting the providers. A decentralized system of communication is not feasible and ideally we could mitigate the risk by restricting both insider and outsider access to data while still maintaining ease of use.

In an ideal world, average users would encrypt all their communication and providers would encrypt and sign all the data they store at all points. Users could then trust providers on the basis of their signatures on every piece of data. However, it is not feasible to rely on average users to verify encrypted messages. Various models have been proposed to solve this problem. In a "Web of Trust" model, a chain of trust must be found between the various parties involved in communication, a difficult and infeasible endeavor. Another option would be a Certificate Authority, but that system requires universally trusted third parties. Additionally, neither the Web of Trust nor the Certificate Authority approach can cope with the threat of an insider attack whereby the provider either chooses or is coerced into issuing a fake certificate of user-key binding and thus allowing a man-in-the-middle attack.

The CONIKS system offers a third approach which combats the issue of a fake certificate by changing the requirements of a certificate. Instead of requiring that a user-key binding be *correct*, we require only that it be *continuous*, meaning that from one point in time to the next, it is verifiable that the user-key binding did not change. We rely on the fact that the first user-key binding was in fact definitionally *correct*. This is the standard practice in systems today. When we see a binding to the username *alice@foo.com* we don't question whether that username belongs to a person named

Alice. We only care that the binding to *alice@foo.com* is the correct binding, i.e. the same binding as before.

In order to ensure continuity we require two things. First, we require that there be no *equivocation*, i.e. that there be exactly one binding per user and that all users who query for *alice@foo.com* get the same binding in response. Second, we require that the user-key binding during epoch $t+1$ be the same as it was during epoch t. In order to verify non-equivocation and continuity, we use a Merkle Tree and respond to lookup requests with a cryptographic commitment of the tree's root and a path from the root to the leaf node. The properties of the Merkle Tree ensure that the tree is valid from one epoch to the next.

However, this second condition of strict continuity is not feasible in the real world. Users need to be able to change their keys at will and in a verifiable way. In order to meet the requirements of *continuity* while still allowing key changes we introduce in CONIKS 2.0 a method of changing user-key bindings by way of a *signed key change*. The procedure is relatively simple. At time $t = 0$ when the user registers a username-key binding, she also publishes a public key. If at any time in the future she wishes to change her binding, she signs a message with the corresponding private key thereby making a verifiable key-change. The CONIKS server will then verify the signature and publish the signed key change. Any user looking up the user and seeing a key change will also verify the key change using the previously published public key. Thus, a username is now bound to a "key blob", a public key used for changing, and a signed key change message when applicable. This new setup introduces a new set of issues which will be addressed below.

The majority of this paper will focus on the changes in the reference implementation made to the original CONIKS from January-April 2015 as part of my independent work.

2. Design Goals

As outlined in Melara et. al. [2] there are six major goals of the CONIKS system.

2.1. Security

G1: Non-equivocation. A provider should not be able to equivocate - that is, a provider, whether by choice, coercion, or infiltration, should not be able to send different versions of a user-key binding without maintaining two versions of the entire chain forever. CONIKS provides a cryptographic proof that ensures that any provider who attempts to equivocate will be caught with high probability. Being that any equivocation will result in irrefutable evidence, no provider will equivocate because of the damage to its reputation.

G2: Key binding consistency. A provider should not be able to change a user's binding without being caught quickly. CONIKS does not prevent a provider from changing a user's binding, but it does alert users when such unauthorized changes happen. A major goal of CONIKS 2.0 is enabling signed and verifiable key changes, thus preventing malicious providers from changing users' bindings without getting caught.

2.2. Privacy

G3: Privacy-preserving consistency proofs. An attacker should not be able to learn information about users in the CONIKS system even with an arbitrary number of consistency proofs. A minor goal of CONIKS 2.0 is allowing users to explicitly disallow public lookups. This will be discussed further below.

G4: Concealed number of users. A provider can insert dummy users in order to hide the exact number of users in the CONIKS system.

2.3. Deployability

G5: Strong security with human-readable names. CONIKS only requires that users know their friends usernames, providing strong security with human-readable names.

G6: Efficient data structure for key directories. CONIKS only incurs a small overhead (proportional to the logarithm of the number of users in the system) and thus is feasible on today's Internet infrastructure. The data transferred is small enough that it can even be handled on mobile phones.

3. CONIKS 2.0

3.1. Changes in the Data Model

Previous versions of CONIKS assumed that the system would be used primarily as a public key store. While we continue to assume that public key stores will be the primary use case for CONIKS, we now remove any assumptions about the form the data will take allowing CONIKS to be used for any application. The reference implementation now refers to "data blobs" instead of public keys. A user has an entry that contains a username, data blob, options, public key for key-change and signed message if applicable.

3.2. Changes in the Security Model

Previous proposals for CONIKS defined *consistency* in the following terms:

An entry shall be considered consistent if it fulfills one of the following three conditions:

1. $\text{Blob}^t = \text{Blob}^{t+1}$
2. There is a signed key change between epoch t and $t+1$

3. There is a signed binding revocation between epoch t and $t+1$

However, the original reference implementation only provided for fully consistent bindings (as per 1.) because there was no mechanism in place for signed key changes. The new CONIKS 2.0 reference implementation uses public key cryptography to allow for signed key changes. The new signed key change capability fully allows for the second and third options. A published change key of all 0s should be considered equivalent to a signed binding revocation. Any point in the future during which a binding is detected that is not all zeroes should be considered a violation of the consistency principle.

4. CONIKS Crypto

4.1. Threat Model

CONIKS defends against attacks that compromise the consistency of a user-data binding. As outlined in the Design Goals, consistency has two parts. First, at any given time, all users looking up a user should receive the same response from the server. We call this **Non-equivocation**. Second, from time t to time $(t+1)$ a binding must either be the same or there must have been a signed change as outlined above in "Changes in the Security Model". We call this **Verifiably Consistent Binding**.

The main threat CONIKS defends against is an insider attack. A server is run by a provider that may be compromised, complicit or coerced into sending a false binding for a given user. A provider may do this in order to execute a man-in-the-middle attack or to deny two parties the ability to communicate securely. By ensuring **Non-equivocation**, a provider cannot send two different bindings on two different requests and thus the risk of such a MITM attack it mitigated.

In the real world, users will want to be able to change data, and so we cannot require that data will always be the same. By allowing data changes, we introduce the risk of a malicious provider changing a users data without authorization. CONIKS 2.0 introduces **Verifiably Consistent Bindings** which prevent providers from changing data without authorization from the user.

The CONIKS security model relies on a principle of "trust on first use" where Bob trusts that the binding for Alice is true the first time he queries the server. In order to ensure consistency, we require that Alice (or some other party) is frequently checking her own binding to ensure consistency. However, at some point in the future, Alice may no longer want to check her binding, and may want to permanently *revoke* the binding associated with her name. To facilitate this behavior, we suggest that a client implementation interpret a data blob of all zeros as a binding revocation. Any change after the "zero-blob" should be considered invalid. Now, when Charlie queries for Alice for the first time, he can look at all previous epochs to see if Alice issued a key revocation. Note that Charlie is not required to do so, but can if he wants to.

4.2. Merkle Tree

A Merkle Tree is a data structure that enables verification of non-equivocation. The structure is organized as a binary tree where each node contains a hash of its two child nodes. Any change in the data structure is reflected in a change to the root node. Any change to the data structure takes time proportional to the logarithm of the number of users. A cryptographic poof of non-equivocation can be provided by sending the entire path from the root to a leaf node. Because of the structure of the Merkle Tree, this path proves that there exists a path from the root to the leaf and that there exists no other such path. The space requirements of this proof is proportional to the logarithm of the number of users, allowing for relatively painless speed and data requirements on today's Internet infrastructure.

In order to ensure consistency, the Merkle Tree's root contains the hash of the root for the previous epoch. By comparing this hash to the previous root, users can ensure that the bindings in the proof are consistent from one epoch to the next. Because the hash is contained in the root, this extra check incurs only constant overhead in both time and space.

4.3. Signed Key Changes

Previous versions of CONIKS only supported initial username-key binding but once a binding was published it could not be revoked or changed. Enabling key changes was a major goal of CONIKS 2.0. Here we deal with the security of the signed key changes.

Signed key changes rely on the principles of asymmetric encryption using protocols like RSA or DSA. A user generates a key pair and keeps one key as the private key used for signing and publishes a public key used for verifying signatures. Because of the cryptographic properties of the algorithms, a signature cannot be faked. By signing the binding change message and publishing a public key alongside that data, all signed key changes can be considered verifiable.

During the development of the original CONIKS reference implementation, code was put in place to use DSA keys as an example of a user-key binding. Although the current implementation uses strings instead of DSA keys for the data, the existing code was repurposed and we now use DSA for signed key changes. Other implementations can use RSA or other asymmetric public key protocols for signing key changes.

Paranoid users can specify that all data changes must be signed in order to be considered valid. When registering, a user publishes a public key alongside her binding. If the user wants to change her binding, she signs a binding change message with the corresponding private key. A user who looks up the user after the binding change can use the published public key to verify the signature.

4.4. The Password Problem

By requiring that all blob changes be signed, we add a constraint on a user that she must always maintain access to her private key. If at any time she loses control of that key (i.e. lost device, file corruption, etc) she will no longer have the ability to change her data. Because requiring users to never lose private keys is sometimes an unrealistic expectation, we allow users to specify whether they will allow unsigned key changes. The current reference client assumes that a user who is making a signed change also wants to restrict changes. We maintain a `allowsUnsignedKeychange` flag that, when set to `FALSE` indicates that a user wants to require that all changes to her account must be signed. Once the `allowsUnsignedKeychange` flag is set to `FALSE`, the user can sign a request to change that flag.

Setting the `allowsUnsignedKeychange` flag to `FALSE` should be done with caution. This setting prevents any kind of provider-initiated data change, for example a password reset. However, leaving the `allowsUnsignedKeychange` flag as `TRUE` limits the security provided by CONIKS. Users and services will have to assess the tradeoffs between the two options.

Note that a client can issue signed key changes, without setting the `allowsUnsignedKeychange` flag to `FALSE`. In that case, when Bob queries for Alice, he can verify any data change with the public key. If for some reason Alice forgets her private key, she can ask the provider to reset her password. Alice should then inform Bob that he should expect to find an unauthorized key change at a specific epoch. When Bob now queries for Alice, he will find that there was an unauthorized key change. He can either reach out to Alice, using some out-of-band communication, to see if she is aware of the change, choose to assume that the change is valid or choose to assume that the change was invalid and ignore it and forgo communicating with Alice using this provider.

If the `allowsUnsignedKeychange` flag is set to `FALSE` and a user finds an unsigned key change, she should report it publicly as a violation. If the `allowsUnsignedKeychange` flag is set to `TRUE`, an unauthorized key change should not be considered a violation. Providers should sign all key changes done on behalf of users with their own public key. Thus, when Bob queries for Alice and expects a signed key change, but the `allowsUnsignedKeychange` flag is set to `TRUE`, he should find either a signed key change from Alice or a signed change from the provider.

If the provider makes a change on behalf of a user, it should alert the user. The user should then immediately generate a new key pair and publish the public key. Any future changes should be signed with the corresponding private key.

5. Other Changes in CONIKS 2.0

There were two small changes made to the original specification in addition to the ability to make signed data changes. First, the previous proposals assumed that the primary purpose of CONIKS would be to store user-key bindings. CONIKS 2.0 makes no assumptions about the type of data in the binding and stores the data as a `String` and renames the field to "`blob`".

Second, each user has a `allowsPublicLookup` flag. Part of CONIKS' security model requires that all lookup requests either receive a proof of a user-data binding or a proof of non-existence. We introduce an option for a user to request that her binding not be publicly viewable except by a whitelist of users. Even when the flag is set to `FALSE`, CONIKS cannot send a proof of non-existence. Instead, CONIKS should send a proof that the user-data binding exists, but not the actual binding.

We leave the implementation of this feature to future work on the project.

6. Related Projects

There are other projects being developed with similar structures and goals. Most noteworthy is a project led by Google called End-to-End. [1] End-to-End uses a similar structure to CONIKS but instead of allowing users to audit the system, End-to-End relies on trusted third parties to act as "monitors". Having a third-party audit the system does add some security, but requires that the public trust that auditor. Considering that there are a limited number of companies with the required capability to constantly verify the system, and that such a third party, like a provider could be corrupted or coerced, we believe that End-to-End and other schema relying on third parties are inadequate.

7. Future Work

There is still much work to be done on CONIKS. First, my branch (mrochlin) of the project must be merged into the soon-to-be-published reference implementation, thus providing anyone access to an implementation of the CONIKS system. The reference implementation will need to be thoroughly tested and made ready for publishing.

The reference implementation serves as a good example of a theoretical CONIKS implementation, but may not be robust enough for very large providers. Further research will need to be done to allow for multiple servers sharing an underlying key store, ensuring reliability and consistency on distributed systems and integrating CONIKS into existing communications infrastructure.

Some of the design decisions were made purely for legacy reasons, and further research should explore alternatives. For example, the decision to use DSA instead of RSA or some other protocol was based on the existence of DSA compatible code in the reference implementation. While DSA and RSA achieve similar security guarantees, research should explore which is more suitable for a system like CONIKS.

Additionally, the reference implementation currently uses Google's protobuffer protocol to send messages between the client and server. While this does make implementing a Java client easier, it is not easily extendible and requires special files and libraries that are not widely distributed. A future implementation may use JSON or some other widely available protocol to pass messages between the client and server.

As mentioned, while some scaffolding is in place to allow users to make their bindings private to a whitelist of users, we have not yet implemented this feature. Because CONIKS cannot simply lie and send a proof of non-existence, further research must explore how to allow for private bindings.

8. Reflections

My work on CONIKS was mainly implementing features as outlined in the Future Work section of the CONIKS proposals. Working off an existing and constantly changing code base presents challenges that many independent work students do not have to face. For example, half way through the semester the topology of the client and server files completely changed. Additionally, keeping the client-server communications schema synced was non-trivial. Luckily, I had great support from Marcela Melara, and such issues were resolved. While the submitted version of this project lives on a different branch from the published reference implementation, we plan to merge my work before the summer begins.

Working on a security project written in Java is no small task, especially because of the complicated way Java deals with keys. Modern versions of Java do have support for DSA and RSA keys, but they use a complex system of key generators, factories, parameters and instances to implement such public key encryption. For example, an instance of a `DSAPublicKey` contains `DSAPublicKeyParams` along with an instance variable *Y* representing the last parameter. Signing and verifying requires a separate Signature class, and generating the keys requires KeyFactory and KeySpec instances. Even generating a key pair is non-trivial.

9. Conclusion

CONIKS provides a new level of security and verifiability in a way that is scalable and useable in the real-world. With the new ability to change keys, CONIKS lives up to its original mission: being a scalable, trustable and easily usable key verification service. The new improvements make the CONIKS prototype completely application agnostic, enabling providers to use the verification service for a wide array of services.

10. Acknowledgements

Special thanks to Professor Ed Felten for advising me on this project. Thanks to Marcela Melara who proposed the original CONIKS system and was indispensable in helping me understand the code base and the project outline and for assisting me and pointing me in the right direction whenever I needed help. Thanks to all my friends who helped proofread this paper, including Jeffery Kessler '15, Tehila Stone '17, Renee Zwillenberg '17, Josh Moskovits '17 and Hillel Friedman '17.

References

[1] Google, "End-to-end," https://github.com/google/end-to-end/wiki.

[2] M. S. Melara *et al.*, "Bringing deployable key transparency to end users," Cryptology ePrint Archive, Report 2014/1004, 2014, http://eprint.iacr.org/.