Hydra: A Trustless Decentralized Digital Identity System

Anish Athalye
aathalye@mit.edu

Ankush Gupta
ankush@mit.edu

Kate Yu
kateyu@mit.edu

Abstract

Current solutions for unified digital identities are insecure and have not been successful in reaching wide adoption. These solutions have several pitfalls due to being centralized and requiring the user to trust a third party. We propose a trustless decentralized digital identity system as a novel solution to the problem of maintaining unified digital identities.

1 Introduction

Over the last 20 years, there has been an explosion in rich, interactive web applications. Many Internet services as electronic mail, social networking, and millions of others, form parts of people's online identities. However, because the Internet was built without a way to know who you are connecting to, every Internet service has come up with a workaround, implementing its own digital identity system. As Kim Cameron wrote in 2005, "today's Internet, absent a native identity layer, is based on a patchwork of identity one-offs" [1].

Since 2005, there has been some improvement in this area. With the advent of OpenID [2], some sites have incorporated support for federated identity management [3], a step toward cohesive digital identity. The scheme allows users to sign in to a single OpenID provider and use that as a proof of identity for all other sites that support logging in with OpenID.

Unfortunately, there are myriad of problems with OpenID, and even though many do support logging in via OpenID, the technology is not generally viewed as a success. OpenID has faced adoption problems partially because all sites want to be OpenID providers and not consumers, because providing login to other sites via OpenID increases the value of having an account on the providing site but accepting OpenIDs does the opposite [4].

OpenID has major security issues. Even though OpenID is described as decentralized [5], it is not truly so. Many OpenID consumers discriminate between sites, only allowing certain OpenID providers. OpenID has availability problems as well: if a users' OpenID provider goes down, the user is locked out of all web accounts that used that login. If the provider goes down permanently, it will be incredibly difficult for users to recover their online identity. Even if a provider stays online, a user has to have absolute trust in their provider. A malicious provider could impersonate the user on every website on which the user logged in via OpenID.

These issues are not inherent to online identity systems. We propose a trustless, truly decentralized digital identity system that overcomes many of the issues and limitations of existing solutions for a unified digital identity.

We outline our general approach to building a trustless decentralized digital identity system. We make heavy use of asymmetric key cryptography, specifically, the RSA algorithm [6].

At the heart of our design, an *identity* is a public key, and a private key is used to generate *proof of ownership* of an identity.

To achieve our goal of true decentralization and fault-tolerance, we build a authenticated data store on top of an untrusted highly-replicated distributed peer-to-peer data structure that provides eventual consistency. We store profile data in this authenticated data store.

To create an identity, a user creates an RSA public/private key pair and publishes the public key to the network. To identify to a service using a public key, a user participates in a challenge response protocol to prove possession of the corresponding private key. The user can publish digitally signed and encrypted profile data to the network such that the information is always available to services that the user chooses to share it with.

Our trustless decentralized digital identity system solves many of the problems present in current solutions, such as trust and availability. Because the service is

truly decentralized with no party controlling any identities, web service operators may be quicker to adopt the technology.

Organization The rest of this paper is organized as follows. In § 2, we describe our peer-to-peer gossip network. In § 3, we describe our eventually consistent authenticated data store and its security guarantees. In § 4, we describe the profile service itself, describing how users create identities, grant and revoke access to services, and participate in a challenge-response protocol to prove ownership of their identity. In § 5, we briefly describe our current implementation of the Hydra protocols. In § 6, we discuss extensions to Hydra and future work. In § 7, we conclude the paper.

2 Gossip network

At the lowest level of our system, we use a connectionless peer-to-peer gossip network that maintains membership information and provides a messaging and broadcast service. For a clean separation of concerns, this layer does not make use of cryptography – security is provided by higher layers.

The gossip protocol implements peer exchange such that nodes can be given the IP address of a single node that is part of the network, and after that, the joining node can automatically discover peers that are a part of the network. This is done through a simple peer exchange protocol where nodes periodically poll known peers for their peer lists. After learning about new peers, the nodes verify that the peers are reachable and add them to their own peer list. Periodically, nodes ping peers, and if no response is received, the peers are marked as dead and the node stops sending them messages.

The protocol also implements a messaging and broadcast service. Nodes using the protocol can send a message to a single peer or broadcast a message to all known peers. Due to the nature of dynamic membership, the messaging protocol does not provide guaranteed message delivery. Fault tolerance in this area is implemented in higher layers.

3 Authenticated data store

Building on the gossip layer, we build an eventually consistent replicated authenticated data store. This data structure provides a globally-shared update-only key-value mapping where keys are RSA public keys and values are arbitrary strings. To store data for a public key, the data has to be signed with the corresponding private key. The system is designed to be update-only, so updates to map keys are timestamped, and newer updates will always supersede old changes. To implement this system, servers discover each other through the gossip layer and periodically participate in an anti-entropy protocol to learn about updates from each other. Currently, this protocol involves each participant sending its full data structure to the other and then performing the necessary checks for correct signature and newer timestamp.

Nodes maintaining this authenticated data store are mutually untrusting parties – security is provided through the use of cryptography.

3.1 Security guarantees

The implementation provides two key security guarantees.

Update-only Honest servers will never accept data for a key unless it is digitally signed, and honest servers can never be fooled into accepting stale data or removing existing data, providing update-only semantics.

Authentication Clients fetching data from the authenticated data store will never accept the data unless it has been digitally signed by the holder of the private key corresponding to the public key.

Due to the nature of having mutually untrusting parties maintaining the data store, it is not possible to protect against denial of service – when communicating with a dishonest server, the server can always choose to not respond to the client. However, because of the way the system is designed, the server cannot lie to the client. Furthermore, the dishonest server cannot disrupt the service through any communication with honest servers.

4 Profile service

The profile service utilizes the authenticated data store to communicate each user's profile information. Users are identified by their public keys, and each public key is associated with a respective blob of information. The profile service provides an interface for managing the contents of this blob. It allows clients to grant and revoke read privileges for other clients, and allows clients who have been granted read permissions to decrypt the data.

4.1 Data structure

The data structure utilized by the profile service is shown in Figure 1.

The `Info` attribute of the data structure is the ciphertext of the user's profile information, encrypted with symmetric key K.

```
{
  Info: enc_K(info),
  SelfKey: P_{PK0}(K_self),
  AuthorizedServices: [
    enc_{self}(PK1),
    enc_{self}(PK2), ...
  ],
  Keys: [
    P_{PK0}(K),
    P_{PK1}(K), ...
  ]
}
```

Figure 1: The structure of a profile.

The `SelfKey` attribute is the ciphertext for a second symmetric key, K_{self}, which has been encrypted using asymmetric encryption using the user's public key. The user can therefore easily determine K_{self} using his private key.

The `AuthorizedServices` attribute is a list in which element corresponds to a service that is authorized to read the user's information. Every element is the ciphertext of an authorized service's public key, encrypted with K_{self}. This allows a user to determine which services are currently able to read the user's data, but means that other services are not able to determine this information.

Similarly to the `AuthorizedServices` attribute, the `Keys` attribute is a list in which every element corresponds to an authorized service. Each element in the list is the ciphertext obtained by asymmetrically encrypting the symmetric key K with the public key ID for an authorized service. An authorized service can now iterate through the `Keys` list until it can determine K using its private key. Using K, the authorized service can decrypt the `Info` attribute. Anybody who possesses the private key corresponding to the entry can modify the profile data to change the `Info` attribute, sign it for use in the authenticated data store, and publish the updated version.

4.2 Creating identities

The core basis to Hydra's identification system is that identities are tied to public keys. In order to create an identity, the client must therefore first generate a new public/private keypair. The profile service also generates two new symmetric keys K and K_{self}. We encrypt an empty string using K as the `Info` attribute, and we encrypt K_{self} using the generated public key for the `SelfKey` attribute. The default value of `AuthorizedServices` is an empty list. Finally, we initialize `Keys` to contain only S encrypted using the newly generated public key.

This new profile can now be modified or published to the authenticated data store as-is.

4.3 Granting and revoking permissions

A user may grant and revoke permission from services to view his or her profile information.

Suppose you have a profile with the `Info` field encrypted with symmetric key K_0. The algorithms by which to grant and revoke permission to view your profile are enumerated in Figures 2 and 3.

Granting Permission to PK_{grant}

1. Decrypt `Selfkey` to obtain K_{self}.
2. Encrypt PK_{grant} with K_{self} and add the result to the `AuthorizedServices` array.
3. Decrypt the first entry in the `Keys` array with your private key to obtain K_0.
4. Encrypt K_0 with PK_{grant} and add the encrypted key to your `Keys` array.
5. Publish the changes to the network.

Figure 2: Algorithm to grant a service permission to view one's profile.

Revoking Permission from PK_{revoke}

1. Decrypt `Selfkey` to obtain K_{self}.
2. Decrypt, find, and remove PK_{revoke} from your `AuthorizedServices` array.
3. Decrypt the first entry in the `Keys` array with your private key to obtain K_0.
4. Decrypt the contents of the `Info` field with K_0 to obtain your decrypted information, I.
5. Generate a new symmetric key, K_1.
6. Encrypt I with K_1 to obtain I_{enc} and replace the `Info` field with I_{enc}.
7. Encrypt K_1 with your own public key and each of the remaining public keys in `AuthorizedServices` and replace the `Keys` array with an array of K_1 encrypted.
8. Publish changes to the network.

Figure 3: Algorithm to revoke permission to view one's profile from a particular service.

4.4 Challenge-response

In addition to providing identity, our system also provides a means of authentication. Using a modified version of the Needham-Schroeder PK protocol for identification [7], two clients may achieve mutual authentication.

Algorithm. Let A and B be two parties that wish to mutually authenticate. Let P_A and P_B denote the public key encryption with A and B's public keys, respectively. A series of messages are sent between the two parties as shown in Figure 4.

$$
\begin{array}{lll}
A \to B: & h(r_1),\ A,\ P_B(r_1, A) & (1) \\
A \leftarrow B: & h(r_2),\ B,\ P_A(r_1, r_2, B) & (2) \\
A \to B: & r_2 & (3)
\end{array}
$$

Figure 4: Needham-Schroeder PK protocol for identification modified for mutual authentication.

In the authentication scheme, r_1 and r_2 are nonces randomly generated by A and B respectively, and $h(r)$ is a *witness* that acts as a zero-knowledge proof of the sender's generation of that nonce.

Upon decrypting the challenge message, B will only continue the protocol iff $h(r_1)$ equals the hash of the decrypted nonce r_1, and the decrypted identifier A equals the identifier sent in the message. If these conditions hold, B will send a similar challenge message back, along with r_1 as proof of B's ownership of its public key. A performs a similar check as well as verifies that r_1 is correct, and returns the decrypted nonce r_2 back to B. Successful completion of this algorithm implies mutual authentication.

5 Implementation

Our system was implemented in roughly 2000 lines of Go code. Our system is highly modular in that the client and server components are agnostic to the implementation of the other, so long as our protocol is followed.

For the cryptography component of our system, we used Go's `crypto` library. For symmetric encryption, we used AES-256 in CBC mode, and for asymmetric encryption, we used RSA-OAEP with 2048-bit keys.

We also wrote command line interfaces by which to generate keys and to perform other client operations, such as publishing updates and granting permission, as well as the challenge-response protocol as described in § 4.4.

6 Future work

As currently implemented, the system functions as a proof-of-concept of a trustless decentralized identity system. Before such a system could be used in the real world, there are some changes that would need to be made.

6.1 Efficient anti-entropy

Currently, when two nodes in the system participate in anti-entropy, the nodes send their full data structure to each other. This is sufficient for a proof-of-concept, but it will not scale to a real-world system. In a real system, there will be millions or billions of identities, and exchanging all that data on every anti-entropy session will be infeasible.

In real-world use, there may be lots of identities, but updates for any given profile will not be that frequent, perhaps on the order of several times a year on average. A scheme to efficiently perform anti-entropy would need to efficiently identify small differences in data stored on nodes. We could do this by storing the data in a Merkle tree and comparing hashes to efficiently identify differences in the tree. This is a common technique in anti-entropy protocols used in eventually-consistent storage systems [8].

6.2 Space-efficient permissions

Currently, our scheme for storing profile data takes $O(P+S)$ space per user, where P is the size of the profile data and S is the number of services the user has authorized. The scheme requires no storage on the services. Ideally, the identity system would only require the network to store $O(P)$ data per user in the network, and individual services could store some additional data to make permissions work properly, and this would result in a huge reduction in data storage required for the servers maintaining the identity system. Currently, we do not know of a way to implement such a system where services themselves are not required to operate online.

6.2.1 Fine-grained permissions

In a real-world system, it could be desirable to share certain parts of profile information with certain services. For example, a user may want to share their full name with one service and their phone number with another. This is not possible in the current implementation of the system. This could be achieved by encrypting the profile data separately for each service, but this would take up $O(P \cdot S)$ space, which is undesirable. It would be nice to have a scheme that supports fine-grained permissions while only using $O(P+S)$ or $O(P)$ space per user.

6.3 Sharding

As the system is currently implemented, every node in the identity system maintains a complete data structure of all profile data. This makes the safety guarantees straightforward to achieve at the cost of requiring every server to store all the data. This works in a proof-of-concept system, but it would not scale to the real world. Ideally, data would be sharded and replicated over the nodes in the system. This could involve switching from a gossip network to a Distributed Hash Table (DHT) protocol like Chord [9] or Kademlia [10]. It is more difficult to achieve our security guarantees when using a DHT because of the types of attacks that can be performed on DHT-based systems [11].

6.4 Adoption

For a new identity system to be used in the real world, users and services would need an easy way to migrate over.

We can simplify adoption by web services by implementing a proxy service that provides a web API to access the network. We note that this service does not have to be trusted, because data is digitally signed. This bridge web service could implement the OpenID API for ease of use.

We can improve the user-friendliness of the identity scheme with web browser extensions that can store RSA key pairs and identify with web sites. This browser extension can be open source, allowing for security validation, such as verifying that the extension does not leak keys.

7 Conclusion

Hydra is an open-sourced system that will allow users to easily utilize a decentralized and secure authentication system to access services on the Internet. Hydra allows these services to easily authenticate users, and allows anyone to participate in the decentralized storage of user profile information. Hydra is resilient in the face of individual node failures, and prevents any party from impersonating another without possessing that party's private key.

By eliminating several of the problems associated with current identity providers, Hydra provides a pathway towards an Internet with a trustless, truly decentralized digital identity system.

References

[1] K. Cameron, "The laws of identity," tech. rep., Microsoft Corporation, May 2005. https://msdn.microsoft.com/en-us/library/ms996456.aspx.

[2] D. Recordon and D. Reed, "Openid 2.0: A platform for user-centric identity management," in *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, (New York, NY, USA), pp. 11–16, ACM, 2006.

[3] S. Shim, G. Bhalla, and V. Pendyala, "Federated identity management," *Computer*, vol. 38, pp. 120–122, Dec. 2005.

[4] D. Obasanjo, "A proposal for social network interoperability via openid." http://www.25hoursaday.com/weblog/2007/08/13/AProposalForSocialNetworkInteroperabilityViaOpenID.aspx, Aug. 2007.

[5] The OpenID Foundation, "What is openid?." http://openid.net/get-an-openid/what-is-openid/.

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[7] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1996.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[10] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 53–65, Springer-Verlag, 2002.

[11] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 261–269, Springer-Verlag, 2002.