Nigori: Storing Secrets in the Cloud

Ben Laurie
(benl@google.com)

May 23, 2010

1 Introduction

Secure login is something we would clearly like, but achieving it practically for the majority users turns out to be surprisingly hard. In particular, we would like it to be possible for a user to log in from anywhere, using nothing except standard software and information that is in his head, without making himself vulnerable to phishing attacks. We, of course, do not exclude more stringent requirements for login, but we wish to satisfy this lowest common denominator.

We start by listing detailed requirements (and non-requirements) for a practical system and then present a system which satisfies those requirements,

2 Requirements

1. The user should not have to remember any more than a single "master" password and a URL.
2. The user should be able to log in to any site from any machine with appropriate software installed.
3. We should not require sites at which the user logs in (relying parties or RPs) to change how users log in. That is, the user should still be able to be logged in with a password.
4. For sites that want to support stronger authentication mechanisms, such as public/private keypairs or other cryptographic mechanisms, this option should be available.
5. When a user logs in at an RP with a password, it should be different from the password used at any other RP.
6. It should be possible to change the password used at any particular RP.
7. The user should not have to trust any single external entity in order to remain secure.
8. The user should have choice over which software implements the system (i.e. we believe in open standards).
9. Attacks on master passwords must not be offline attacks, except for attacks by any entity (or group of entities) which are providing the storage service. Clearly, they cannot be prevented from mounting an "offline" attack by simulating the user in a closed network environment.
10. Modification of stored secrets should be detectable.

3 Non-requirements

1. A standard browser or other client as it exists today - we explicitly do not exclude the possibility of a strong solution requiring modification to existing browsers and clients. Nevertheless, we will present a variant of the system which should work in existing browsers.

4 Consequences of the Requirements

It seems clear that in order to satisfy the requirements there must be a place or places where a user's secrets (passwords, private keys, etc) are stored. These secret stores must not, however, be able to access those secrets.

This means the secrets need to be encrypted in such a way that the store has no access to the encryption key.

Because attacks on the master password should not be offline, the user must authenticate to the store before retrieving the secrets, and that authentication must be derived from the master password.

This, in turn, means that the user cannot authenticate by simply providing his password to the store, as is currently standard with web authentication.

The secret stores must also be accessible from anywhere - that is, they need to be "in the cloud".

Because there should be no offline attacks available, and modifications should be detectable, only the legitimate user should ever be able to see the secrets. This means that the secrets should be retrieved over a secure channel, otherwise a man-in-the-middle or a snooper would be able to take the encrypted secret and try master passwords until the integrity check succeeded. We do not discuss in any detail what secure channel is used because these already exist, for example in the form of TLS or OpenSSH connections between client and server.

5 Details of the System

Here we describe the Nigori protocols in some detail (precise encodings and parameters, however, are described in the Nigori protocol document).

All connections are assumed to be over a secure channel which can prevent eavesdropping and man-in-the-middle attacks, for example a TLS connection authenticated by a server certificate.

5.1 Key Derivation

We allow two mechanisms for key derivation - the first reduces the level of trust the user must have in the service providers, but provides no mechanisms for recovery if the user forgets his password (other than, obviously, the user providing a way to recover the password itself, such as, for example, writing it down, or telling it to a trusted friend). The second allows for recovery in the event that the user forgets the password, in exchange for greater exposure to service providers. We call these unassisted and assisted password-based key derivation, respectively.

In both cases, three keys are derived, K_{user}, used to authenticate the user to service providers, K_{enc}, used to encrypt secrets and K_{mac}, used to check the integrity of encrypted secrets (that is, to allow the detection of accidental or malicious tampering).

5.1.1 Unassisted Password-based Key Derivation

In the unassisted mode, K_{user}, K_{enc} and K_{mac} are all derived from the master password in such a way that they are not derivable from each other.

For example, in the current protocol, we use PBKDF2 [Kaliski, B (2000), "RFC 2898: PKCS #5: Password-Based Cryptography Specification Version 2.0", http://tools.ietf.org/html/rfc2898] with a different count for each secret. Note that the current protocol does not include the server name in these, because it is instead included elsewhere in the authentication mechanism, but for other mechanisms it might be more appropriate to instead include it in key derivation.

5.1.2 Assisted Password-based Key Derivation

In the assisted mode, first we retrieve a key K_{master} for the user from some server other than the storage server (we could call this the recovery server). It is up to that server to decide how to create this key, but it must be able to recreate the same key even after the user has forgotten his password for the recovery server (or whatever other means he uses to log in to it). For example, the recovery server could derive K_{master} from the hash of the user's password that it presumably stores to use in password verification.

The client then derives K_{enc} and K_{mac} from K_{master}, and K_{user} from the user's master password (which may be the same as the password he uses to log in to the recovery server). Note that even if both passwords are the same, an honest recovery server cannot derive K_{user} from data it has stored, because it should not store the password, just something derived from it in order to verify the user.

Recovery from a forgotten password in this case is possible - the user performs account recovery in the usual way (e.g. through a helpdesk, via email or SMS, by proving knowledge of various other secrets, etc.) at the storage and/or recovery servers. The recovery server then provides the old K_{master} (we recommend that it should change after account recovery) and this the client can still decrypt the user's secrets - and re-encrypt with the new K_{enc} and K_{mac}.

5.2 Storage

Now that we've dealt with the derivation of the three keys, storage and retrieval of secrets is relatively straightforward.

Firstly, the user authenticates to the storage server using K_{user}, then either sends or retrieves secrets which are encrypted and authenticated with K_{enc} and K_{mac}.

There are a wide variety of ways the user could authenticate using K_{user}, but we have chosen to use a DSA signature (with K_{user} as the private key) over the username, the server name, a timestamp and a nonce. The server checks that all of these are valid, that the timestamp is recent, that the particular timestamp/nonce combination has never been seen before and that the signature is valid. Note that the timestamp is needed to limit the size of the database used to prevent replays.

The advantage of a DSA signature for this purpose is that it does not give an attacker who has access to the communications (for example, one who has taken over the storage server, or a storage server that is evil) any kind of password equivalent.

Regardless of authentication mechanism used, because K_{enc} cannot be derived from K_{user} (other than through a brute force attack on the user's password), the storage server is unable to decrypt the user's secrets.

5.3 Split Storage

Once all of the above is in place, the user could choose to have his client store his secrets split across multiple servers, such that to even mount an attack on the secrets multiple storage servers would have to collude (note, however, that this is likely to be a poor defence unless he uses different passwords at different servers).

In addition to splitting for security reasons, the user might choose to use multiple servers for resilience.

When splitting for security, a threshold encryption scheme could be used, such as Shamir's secret splitting scheme [Shamir, Adi (1979), "How to share a secret", Communications of the ACM 22 (11): 612613]. When splitting for resilience obviously each server could store the whole of the secret, but nevertheless the user might choose to use a split to mitigate the risks involved (for example, a user who splits for resilience is quite likely to do so because he runs his own server, but is not confident of 100% uptime and so wants to use the servers of friends as backups - but does not want to trust or burden his friends with the whole of his secrets).

5.4 Details of Storage

If we are to achieve our requirements, we must specify not only how we protect our secrets, but exactly what is stored, in order to allow interoperable implementations. Therefore the detailed protocol specifies bit-level formats for the various types of secrets (at this time, these are passwords [i.e. random blobs], RSA private keys and the locations of split storage servers).

Each secret is stored against a lookup value (I hesitate to use the traditional word "key" at this point!) which is the encryption of the information needed to use the resource (e.g. the URL at which the password is entered) and its type.

We also store all the old values of encrypted secrets in case of failed password change attempts, disaster recovery at the RPs and the like which might require a fallback to an older version of the secret. Obviously it would make sense to expire old values eventually.

Finally the storage servers should be prepared to list the (encrypted) lookup values so that users can manually deal with issues like sites which change their name or ownership, or sites where login is available at multiple URLs.

6 Variations on the Theme

6.1 Unmodified Client

One variation we have considered is the situation where the user has only an existing browser, with no client-side support for the protocol. In this case some minor modifications of key derivation details, combined with some standard Javascript and an OpenID or OAuth-like flow of redirects and UI could be used to retrieve and exercise the secrets.

In particular, we would operate in the assisted mode and K_{master} would include a component that was derived from the name of the RP. When the user

arrived at the RP it would initiate a flow like OpenID or OAuth[1], redirecting the user to their IdP. The IdP would supply K_{master}, which would be defended against manipulation by the RP because a dishonest RP supplying the wrong RP name would then not end up being redirected to at the end of the redirection chain. Instead of returning directly to the RP, the IdP would instead bounce via the storage server, which would add the appropriate encrypted secret to the redirection URL. Finally, the user's browser would end up back at the RP with K_{master} and the encrypted secret, which a standard piece of Javascript[2] served by the RP would decrypt and enter into a password box.

The great advantage of this scheme over the standard OpenID flow is that the RP would not have to make any modification to their login flow other than including the standard Javascript for doing the redirection and decryption. The end result would be a password typed in as if by the user.

Note that a dishonest RP would only be able to retrieve the user's password for itself, which is not an interesting attack.

Also note that, because of this, if the URL becomes too long because of the size of encrypted secrets and so forth, the IdP and storage server could instead hand short codes to the RP which it could then use to retrieve the full versions to be processed by the Javascript.

6.2 Granting access via OAuth

If instead of K_{enc}, which is the same across all secrets, we used a value, K_v, derived from both the user's password and the lookup value, then protocols such as OAuth could be used to grant access to individual secrets. There are two variants of this.

6.2.1 Variant 1

The service provider (SP) is granted access to the encrypted key and is also given the value of K_v so it can decrypt it (and so the storage server cannot).

6.2.2 Variant 2

The SP is granted access and the storage server decrypts the secret for it. There seems to be little value in this variant, but we mention it for completeness.

6.3 Exercising the Secret on Behalf of the User

In this variant, the user may be offline and so unable to use the secret via his client, in which case an OAuth grant could be made as above, but rather than the SP getting access to the secret itself, yet another service uses the key on behalf of the user. The advantage of this variant is that the user could grant SPs it doesn't much trust access to, for example, signing keys and could later revoke that access without necessarily revoking the keys.

[1]Both protocols work by a series of HTTP redirects. Each one takes the user to the next actor in the protocol, who may then interact with the user, or may directly take action purely based on the redirected URL, and then in its turn redirects the user's browser, eventually ending up back at the RP with retrieved information in the URL of the final redirect.

[2]By "standard" we mean that it would be largely identical at all RPs and could therefore be easily included as a static modification to the login page.

7 Conclusion

We believe we have presented a protocol which achieves all the goals stated at the outset of this document, and which is also capable of extension to solve a number of previously unsolved related problems.

Once such a mechanism is in place it becomes feasible for users to finally heed the impractical advice they have long been given: use a different password at each site to avoid phishing. Furthermore, it allows those passwords to be entirely random, and therefore strong.

Furthermore, the system can also be used to gradually move away from password-based authentication to stronger mechanisms, such as client certificates or selective disclosure [Laurie, Ben (2007), "Selective Disclosure", http://www.links.org/files/selective-disclosure.pdf].

Finally, the system can be incorporated into most sites with almost no modification, if so desired, whilst allowing a smooth transition to more advanced mechanisms.