friend.me: A private social networking webapp

Alin Tomescu
`alinush@mit.edu`

Prashant Vasudevan
`prashvas@mit.edu`

Sachin Shinde
`shinde@mit.edu`

December 12, 2014

Abstract

We design and implement *friend.me*, a web application for online social networking that provides strong guarantees about user data confidentiality and integrity on the untrusted server. We use the Meteor framework to implement the web application due to its clean separation of code and data. We use ideas from Frientegrity[3] in the design of *friend.me*, but we achieve weaker security guarantees in our implementation. We design a secure *access revocation* scheme that offers some degree of *forward secrecy* for user data: Alice's removed friends are *cryptographically* denied access to Alice's new content, such as pictures and posts.

1 Introduction

friend.me is an online social networking web application similar to Facebook[1]. *friend.me* offers a subset of Facebook features while preserving user data confidentiality and integrity on the server. Users use their browser to access the *friend.me* service, exposed as a web application at `https://friend.me`. The browser fetches *client-side code* (CSS, HTML and JavaScript), authenticated by an in-browser extension (implemented in previous work [7]) so as to prevent a malicious *friend.me* server from serving malicious client-side code that leaks user credentials. The client-side code is verified independently and, as a result, is trusted by the user.

2 Threat model

2.1 Untrusted *friend.me* server

The *friend.me* server is *untrusted*. We only rely on the server for storing encrypted and authenticated data in a database. In that sense, the server can mount DoS attacks, but we believe this to be a non-problem, because such behaviour would go against the financial incentives of a service provider. We **do not** trust the server with users' encryption keys. Such keys are never sent to the server. The server can attempt to create ciphertexts, tamper with ciphertexts, delete ciphertexts or replay old ciphertexts. The server can attempt to replay old content to a user.

2.2 Untrusted non-friends

The *friend.me* users themselves can attempt to access data they do not have access to, such as: profiles of non-friends, pictures of non-friends, conversations they are not part of, posts destined to a group they are not part of, etc. In that sense, with respect to a user u and some content c of u there exists a set of untrusted users s that are cryptographically denied access to content c by *friend.me*. Note that friends, or users which are given access to content, are trusted not to give away encryption keys for that content to the server.

2.3 Trusted friends

Friends of a user Alice are trusted by Alice to correctly execute the *friend.me* protocol. More on this in later sections (verifying hashes on shared content such as conversations, signing hashes on Merkle trees and hash chains).

3 Trusted computing base

A *friend.me* user has to trust:

1. **The *friend.me* client-side code**, to not leak secrets to the server or to other parties outside the browser.

2. **The browser's JavaScript engine**, to correctly execute and enforce browser policies. For instance, the browser cannot allow other webpages to read *friend.me* JavaScript state, like secret keys (i.e. same origin policy has to be enforced).
3. **The Meteor web framework**, to cleanly separate server code from client code. In one extreme example, suppose Meteor caches local client-side JavaScript variables on the server and hence leaks secret keys stored in those variables. Also, Meteor is trusted to correctly *compile* client-side code and not insert malicious statements (like sending keys over to an adversary).
4. **The JavaScript libraries used in the client-side code**, such as encryption libraries. Crypto libraries are trusted to be *correctly implemented* and *side-channel attack-free.* The source of randomness in JavaScript client code is trusted. Other libraries like jQuery are trusted to be free of XSS attacks which would allow an attacker to steal a user's keys.
5. **SSL/TLS**, for authentication, secrecy and integrity of the communication channel. We need secrecy during user authentication by the server which is done by traditional means (hashed passwords and session IDs). We need integrity to prevent network attackers from tampering with in-flight ciphertexts and mounting DoS attacks (such as modifying in-flight user messages and rendering them undecipherable later).
6. **A source of time**, such as an NTP server. The users need a reliable source of time to include in their encrypted and authenticated content (such as photos or messages), so that they can later assess the freshness of this content when they are presented back with it.

4 Security goals

4.1 Secrecy of user data

All user-generated content is encrypted and kept secret from the server, from non-friends, and from friends who are not on the *access control list* (ACL) associated with that content.

friend.me protects users' private data confidentiality using symmetric key encryption and offers *chosen-ciphertext attack* (CCA) security for all ciphertexts.

friend.me also *cryptographically prevents* friends who have been given and later denied access to some user's content (like an album or a conversation) from reading *future* updates to that content (like new photos or new messages). Similary, if Alice defriends Bob, then Bob loses the ability to decrypt any future content Alice might post (on her wall or albums).

4.2 Integrity of user data

We want to prevent the server from assembling ciphertexts together and presenting arbitrary views of a user's content to a visitor. If the client-side code only verifies the integrity of individual ciphertexts but not the integrity of how those ciphertexts are linked to one another, such arbitrary views of content will look authentic even though they are not.

For example, we want to prevent a malicious web server from taking messages from one private conversation and putting them in another conversations. We use *Merkle Trees*[4] on *composite content* such as photo albums and *hash chains* on walls and conversations.

Note that we only provide per object integrity and do not provide integrity over all objects simultaneously: the server can pick and choose which objects it presents to a user. An object can be a wall, an album or a conversation.

4.3 Freshness of user data

In the presence of persistent client-side storage (such as HTML5 local storage) we provide freshness guarantees for walls, albums and conversations, as well as for the ACLs associated with them. As users create their own content (albums) or contribute to shared content (walls, conversations), *root hashes* are remembered on the client and prevent the server from showing old content to a user. However, the server can still refuse to present updates by other users to Alice. For instance, given a conversation $c = m_1, m_2, \ldots m_n$ of n messages that Alice is part of, the server *can only fork Alice by refusing to present all messages* $m_k, m_{k+1} \ldots m_n$. If Alice then posts a new $(k+1)$th message after being forked by the server, then the server can never show messages $m_{k+1} \ldots m_n$ to Alice or she will detect that she's been forked by noticing that her $(k+1)$th message is not cryptographically linked to m_{k+1}.

Note that freshness guarantees cannot be provided without client-storage unless the user is presented with something he can manually verify. Thus, if a user clears his HTML5 local storage then the server can present any version of an object to the user.

5 Design

We implemented *friend.me* as a Meteor 1.0 web application using JavaScript and the Stanford JavaScript Crypto Library (SJCL), ellipticjs, and CryptoJS.

5.1 Users and keys

Since users need to encrypt their content, share it with their friends and possibly revoke some of their friends' access, this implies a user will have more than one cryptographic key, which means these extra keys will have to be encrypted and *authenticated* on the server.

When a user Alice signs up, she chooses her password and that password is never sent to the server. The server only gets a *bcrypt* hash that Meteor uses as a user authentication token. When Alice logs in, the JavaScript client code derives a symmetric $masterKey$ from her password as $masterKey = PBKDF2(password, salt, iterations)$, where the salt is stored on the server. The $masterKey$ is used to encrypt and authenticate an *asymmetric key pair* (ED25519 in our case) denoted by $asymKeyPair$. Alice will need this keypair for signing certain operations.

To befriend Bob, Alice sends him a friend request that she signs. This is to prevent controversial providers like LinkedIn which apparently send friend requests on behalf of their users but without their users' knowledge or approval. Also, Alice obtains a shared symmetric $friendKey$ with Bob after passing a shared key obtained from Elliptic Curve Di ffie-Hellman (ECDH) through a key derivation function (KDF). We assume Alice and Bob have confirmed each other's public keys in an out-of-band fashion.

When Bob accepts the requests, he obtains the same symmetric $friendKey$ after performing the ECDH key exchange and applying the same KDF to the shared ECDH key.

5.2 ACLs

The $friendKeys$ could be used to share content between Alice and Bob but that would be very inefficient: Alice would have to encrypt her content (many albums, wall posts, conversations, etc.) for all of her friends each time she friends somebody and each time she creates new content.

Instead, each object (wall, album, conversation) has a symmetric $contentKey$ generated for it by the owner of the object, and *Access Control List (ACL) trees* (similar to the ones used in Frientegrity[3]) are used to provide sharers access to an object's $contentKey$.

If Alice defriends Bob, then a new $contentKey$ is generated for each object Alice owned and shared with Bob, so as to cryptographically prevent Bob from reading future updates to that object (such as viewing new pictures in an album). This new $contentKey$ is stored in the ACL tree's root, and the path from Bob to the root is encrypted under different keys such that the new key becomes inaccessible for him. Bob proceeds identically with respect to Alice once he is defriended.

ACL trees are "Merkelized" for integrity and their root hash is included in each object update operation, to prevent the server from presenting an older ACL with a newer piece of content. For instance, if Alice posts a wall post to Bob, then she will also include the root hash of the ACL tree in her signed wall post so that users who post other messages after her are guaranteed to pick up the right ACL version.

5.3 Hash chains

We use *hash chains* to provide integrity and freshness for walls and conversations. A conversation is a chain of *quacks* (we use this as a pseudonym for *"messages"* in a conversation and *"wall posts"* in a wall). Each quack contains a signature on its hash and on the previous quack. Thus, all quacks form a hash chain ending in the last quack. The first quack uses the conversation itself as the previous quack, thereby *binding* a quack chain to its conversation. Each conversation contains a unique *client-generated* ID so that quack chains bound to seemingly identical conversations (same owner and title) cannot be swapped between each other.

When a conversation is started by its owner, the owner posts the initial quack (i.e. the first message in the conversation), signing the conversation and the quack and binding them together. At this point, the owner computes the *trusted hash* of the conversation as the hash of the last quack and can verify that new messages posted by other users link back to this quack, upgrading his trusted hash to the latest one. Note that the hash of the last quack depends on the hash of all the quacks before it.

When they first access a conversation, users who are on the conversation's ACL will have to get the hash of the first quack from the server, verify it, and then verify that the rest of the quacks link back to this hash. Then they can upgrade their trusted hash to the latest one and use this trusted hash for verifying new updates. Note that the server can choose to not present the last k quacks in the conversation to a user Bob, but if Bob posts a new quack, thereby forking the hash chain, the server can never present those last k quacks to Bob as they will not verify against Bob's latest quack.

5.4 Merkle trees

For albums, we want to prevent the server not only from modifying individual pictures, which is easily achieved using message authentication codes or authenticated encryption modes, but also from moving pictures in between albums and from not presenting all pictures to a user.

We use Merkle trees for albums, as opposed to hash chains, because they make verification of an individual picture in an album of n pictures only $O(log(n))$ in cost. If we used a hash chain, we would've incurred an $O(n)$ cost, depending on where in the chain the picture is.

As opposed to walls and conversations, only owners can modify an album. As a result, they always have the latest root hash of the album's Merkle tree cached. The difficulty lies in updating the other user's root hash for their friends' albums. Frientegrity solves this problem using history trees which allow users to collaboratively verify that a new Merkle tree hash is linked to a previously trusted Merkle tree hash. We solve it by checking that the version number has increased in the root of the tree. In our case, the complexity of history trees is unjustified in this simpler scenario where only the trusted owner is updating the album.

Unfortunately, neither Merkle trees nor history trees solve the freshness problem: the server can always present an older version of the tree to the album viewer than the version the owner has (but not older than what the viewing user has cached locally).

6 Future work

- Comment threads, likes, friends of friends, and many other social networking features.
- Concurrency issues, both across clients of the same user and clients of different users.

7 Related work

7.1 Frientegrity

Frientegrity[3] is an existing framework for online social networking implemented as a Java client and server. Frientegrity does not go the extra mile of implementing a web application and they do not deal with **verification** of the client-side JavaScript Frientegrity code. This code would need to be authenticated so as to ensure it will not leak a user's secrets to an untrusted party.

7.2 Mylar

Mylar[7] is an existing framework for building secure web applications that stores all key material on the client, making it safe to outsource encrypted user data to an untrusted server. However, it is unclear how Mylar would deal with access revocation when a user Alice defriends Bob. *friend.me*'s goal is to cryptographically prevent Bob from interacting with any new content Alice might post in the future.

7.3 CryptDB

CryptDB[6] is a database server that performs SQL queries over encrypted data by leveraging various encryption schemes that allow computations on the ciphertext. However, CryptDB's architecture relies on a proxy that rewrites SQL queries received from a web application. Consequently, the proxy has to store key material which defeats our goal of never allowing a key to leave the user's trusted machine (see 4.1).

References

[1] Facebook. `https://www.facebook.com`. Accessed November 5th, 2014.

[2] Aris Anagnostopoulos, MichaelT. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In GeorgeI. Davida and Yair Frankel, editors, *Information Security*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393. Springer Berlin Heidelberg, 2001.

[3] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social networking with frientegrity: Privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association.

[4] R.C. Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569.

[5] Colin Percival. Stronger key derivation via sequential memory-hard functions, 2009.

[6] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[7] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 157–172, Berkeley, CA, USA, 2014. USENIX Association.