PreVeil E2EE Email: Security Review

Severyn Kozak, Max Murin, Wendy Wei

6.857, Spring '17

1 Overview

We conducted a security review of the client software for PreVeil, an end-to-end encrypted email service currently in its beta phase. PreVeil packages strong security features, like true end-to-end encryption, private keys instead of passwords, and decentralized key recovery, into a polished and smooth user experience (traditionally a big obstacle to the widespread adoption of secure email). We present a high-level overview of how PreVeil works and its key features (section 2), the threat model it operates under (3), and finally thoroughly evaluate the security of the critical parts of the client codebase (4).

2 How PreVeil works

PreVeil is built around public-key cryptography. The gist is that every user has a private/public key pair – the public key is made public to all other PreVeil users, and the private key is kept secret on the user's machine. To send someone an email, you encrypt it under their public key to ensure that only they can then decrypt it with their secret key. You can access your PreVeil emails on multiple devices by securely transferring your private key, and you can appoint so-called approval groups to securely recover your private key should you lose it.

2.1 Getting started

To get started, you download and install the PreVeil client software and register an @preveil.com account. On account creation, you'll generate a public/private key-pair. The public key gets sent to the server along with your username and other account metadata, whereas the private key gets stored locally (where it should remain).

2.2 Sending/receiving emails

If user A wishes to send user B an email, A first generates a random symmetric encryption key K. A encrypts M under K to produce C_M, and then encrypts K under B's public key PK_B to produce C_K. Finally, A produces a signature σ of C_M using their secret key SK_A. A then sends (C_M, C_K, σ) to the PreVeil server.

To retrieve A's email, B downloads (C_M, C_K, σ) from the server and reverses the above process. They first retrieve A's public key and use it to verify the signature σ. They use their private key to decrypt C_K and thus retrieve K, which they then use to decrypt C_M and retrieve M.

2.3 Key recovery

If a user loses their private key, they're effectively totally locked out of their account. There's no equivalent of a "password reset" since the PreVeil server doesn't know anything about users' private keys. PreVeil implements secure key recovery through Shamir Secret Sharing, which distributes

"shards" of the key to an approval group (of configurable size) – a small group of other users that the account owner trusts. To recover their key, the account owner asks their approval group to send them the key shards, a strict number of which must be collected before recovery can occur (with any fewer shards, the key is mathematically underspecified and simply can't be recovered). Thus, individual shards don't reveal anything about the key, and can thus securely be distributed to individual users.

2.4 Adding new devices

Users will often want to access their inboxes on multiple devices (their home computer, laptop, mobile phone, etc.) To do so, they need to securely transfer their private key to new devices, which is executed through a secure channel brokered by the PreVeil server.

3 The PreVeil Threat Model

In the PreVeil threat model, the centralized server is completely untrusted and is modeled as a passive attacker. That is, an adversary is assumed to have read-only access to the server – they can read any data on it (or passing through it) at any time. Thus, it's critical that no sensitive data, like unencrypted emails or even more important, users' private keys, gets sent to it.

Active attackers, on the other hand, are out of scope. An active attacker is assumed to actively try to sabotage the user, which they might accomplish by deviating from PreVeil protocol, sending users a malicious software update that uploads their private key to an attacker-controlled server, or simply changing every public key in the PreVeil system to the attacker's (which would allow them to decrypt any emails that get sent in the future). For obvious reasons, active attacks are very difficult to guard against, but the PreVeil team is working on features that make them more difficult to mount.

This security review is restricted to verifying the correct functionality of the client software. Auditing server software is unnecessary for our purposes because the server is untrusted in the first place, which means we only need to verify that the client is not susceptible to attack and doesn't send any sensitive data to the server. We assume the correct functionality of technology that PreVeil is built on, like the libnacl cryptographic library and SSL, and also that the user's machine is malware free.

4 Security Review

Here, we thoroughly explore each of PreVeil's major components (sending emails, key management, etc.) and document the code pathways involved, and why we deemed them secure.

4.1 Account creation

The primary concern with account creation is that private/public keys are randomly generated, and that only the public key gets transmitted to the server while the private key remains on the user's computer (we explore this more extensively in the next section, **Key Management**). Account creation is initiated by POSTing a username and display name to the `/users` endpoint on the backend. The user then "claims" and sets up this account by querying the `setupNewAccount()` route on `crypto_server.py`, which defers to `crypto_helpers.claimAndSetupNewAccount()`. Here, a private key is randomly generated with `crypto_helpers.keys.RawPrivateKey()` and written to local storage (`crypto_helpers/crypto_helpers.py:92`), whereas the corresponding public key is sent to the PreVeil server (`crypto_helpers/crypto_helpers.py:66`). The private key doesn't leave the user's computer, so account creation is secure, and from here on it's up to ambient key management to ensure that the private key never leaves.

4.2 Key management

Encryption, decryption, signing, signature verification, and all other cryptographic operations are performed by `crypto_server.py`, which thus is the only part of the codebase that directly handles the user's private and public keys. Private keys can only be accessed through `CryptoUserData.getPrivateKey()` (defined in `crypto_helpers/user.py`), and we manually assessed its usage throughout `crypto_server.py` (15 occurrences at the time of writing) and verified that private keys never left it. Time permitting, a rigorous practical text (like scanning all outbound PreVeil requests for the private key strong) might be an interesting way of confirming that the private key is never accidentally sent out.

4.3 Sending emails

Emails are composed in the PreVeil frontend and then posted to the `crypto_server` daemon at `/put/account/< sender_id>/message`. The request gets routed to the `sendEmail()` function in `crypto_server.py`, which initializes an `EmailSender` for the sender and uses it to send the emails through `EmailSender.sendEmails()`. `sendEmails()` constructs a MIME email object through `crypto_helpers.email_util.buildEmails()`, which it then appends to the local inbox and finally sends to the SMTP server running on port 4001. Even though sending an email through PreVeil only involves uploading it to the `/storage/` endpoint of the server's API, a full-blown local SMTP server is necessary because it allows PreVeil to be used through Outlook, Apple Mail, and other Mail User Agents (MUAs), which expect to send emails to an SMTP server. The SMTP server, implemented in `smtp_server.py`, bundles the email into a `PreparedMessage` (defined in `util.py`), which is what actually handles the encryption in its constructor. `PreparedMessage` first calls `fetchOpaqueKey()` to generate a random key to symmetrically encrypt the email, which it then encrypts with the receiving user's public key through `userEncryptAPI()`. It then separately encrypts the message body and attachments through `self._encryptBlock()`; some parts of the email metadata (subject, recipient, etc.) is private, while others are public (like the message ID and the In-Reply-To header). The `PreparedMessage` is then passed to `sendMessage()`, which uploads it to the server through `PreparedMessage.upload()`; metadata is posted to the server's `/mail/<recipient_cid>` endpoint separately.

4.4 Receiving emails

Assessing the security of receiving emails isn't a huge concern because it simply involves downloading new emails from the server and decrypting them correctly, but we at least want to verify that email signatures are checked. The email-receiving pipeline begins in the `doUpdate()` function in `imap_server.py`; `doUpdate()` immediately calls `postlord.py`'s `implUpdate()`, which fetches "update" metadata from the PreVeil server through `_fetchUpdates()` – this informs it of whether any new emails have arrived in the user's inbox. `_fetchUpdates()` then defers to `_writeUpdateToLocalStore()`, which decrypts the emails in `_reconstructEmail()`. `_reconstructEmail()` verifies the email signature with the following block of code:

```
v = util.userVerifyAPI(sender, signing_key_version, canonical_msg_str,
        signature, doas=user_id, is_text=True)
```

4.5 Approval groups (key recovery)

Approval groups are Preveil's solution to account recovery. If a user loses their private key, it is not possible to recreate that key from data stored on the server. Thus, a user can designate a set of other Preveil users to hold shared secrets, or shards, that combine to form the private key. These shards are created with SSSA, a public library for running Shamir's Secret Sharing in Python. This part of Preveil is an interesting place to look for vulnerabilities, because if any part of this system is subtly wrong, then the server can gain access to a user's private key.

At a high level, the approval group process works as follows. A user, henceforth the recovering user, uses Shamir's algorithm to split their private key into shards. Then, each shard is encrypted with the desired user's public key, and sent to the server. Then, when the key needs to be recovered, the recovering user and the approving user use the WebSocket protocol, relayed by the server, to communicate. The approving user retrieves the shard from the server, decrypts it using their private key, and encrypts it using a one-time key for that session. Once the recovering user receives all the required shards, that user can recover the secret with Shamir's algorithm and regain their private key.

Approval group creation is handled by the `/post/account/<user_id>/approval` route in `crypto_server`. This route requires a json request with properties `required_users`, `optional_users`, and `optionals_required`. This route takes the private key, and uses the class `ApprovalGroup` to generate shards with Shamir's Secret Sharing. First, the key is split into a number of shards equal to the number of required users, plus one extra if there are any optional users, so that every one of these shards must be known to recover the secret. Then, the extra shard is split again over all the optional users, only requiring `optional_users` users. For each of these users, the system gets the public key from the server, and encrypts that user's shard using their public key. Then, these encrypted shards are sent to the server at `https:// collections . preveil .com/users/approvers` using the `APIClientCrypto` class, which signs messages with the user's PreVeil private key.

Since the URL begins with https, these groups are sent to the server over TLS, which verifies the identity of the server using certificates. The user verifies themselves with their PreVeil key. TLS will make the communication private, so that an attacker who overhears the message cannot learn anything about that user's approval groups. This appears secure from network attackers, but this approval group behavior is notably different from Preveil's online explanation at https://www.preveil.com/technology/, which claims that the user's private key is encrypted with a symmetric recovery key, and that this recovery key is then split into shards.

Since PreVeil is an end-to-end system, we must ensure that the PreVeil server does not gain any information about the user's private key from these shards. Since they are encrypted with the approving users' public keys, the server does not gain any information about the key. The server does, however, know what users are part of the approval group, and so an attacker who can read data on the server can learn this information and attempt to use social engineering to retrieve the user's private key.

When a user needs to retrieve the shards from other users, the user cannot interact with the server in the same way as when they are depositing the shards, because they do not have their PreVeil private key with which to sign messages. Instead, the user uses WebSockets over TLS in the `get/account/<getter_id>/shard/<sender_id>/get` route in `crypto_server`. The approving user `sender_id` must simultaneously use the `/get/account/<sender_id>/shard/<getter_id>/send` route on their own version of `crypto_server`, which also communicates with the server using WebSockets over TLS. The server apparently acts as a relay between the two users, taking messages sent by one user and sending them to the other. The two users must communicate through other channels to start the approval process.

When both users are connected, they generate one-time use public/secret keypairs, and send each other the hash of their public key. After both parties receive the hashes, they then send the actual public keys over the same channel, and ensure that they match the hashed values. We are not sure why the system uses hashes in this way; the system claims that this process helps with man-in-the-middle attacks, but it seems to us that a man in the middle who can intercept messages and replace them can simply send their own hash and public key, instead of just their own public key.

Then, the approving user requests the corresponding shard from the server at `https:// collections . preveil .com/users/approvers/shard`. Similarly to the creation of the approval group, messages to the server are signed by the approving user's secret key, and since they are sent over TLS, they are secure under our assumptions. The approving user decrypts the shard, then encrypts it with the one-time secret key for this transaction, and sends it over the WebSocket. The recovering user then decrypts the shard, which they then store locally, and end the transaction.

Once every shard is gathered, they can be restored by SSSA to the full public key. Each approving user only ever sees one shard, so they cannot gain access to the public key without conspiring. Since the approval group model assumes that the approving users are trusted, this vulnerability is out of scope.

4.6 Adding Devices

A single user can use PreVeil on multiple devices. Each device must contain the user's private key in order to use PreVeil. If at least one device has the user's private key installed, then during installation on a new device, the process of key transfer will send a copy of the private key from the old device to the new device.

Key transfer uses the same WebSockets over SSL/TLS protocol that is used for sending shards in approval groups. WebSockets allows for bi-directional messages between the client and the server. SSL/TLS ensures that the server is trusted through certificates and that the communication is authenticated using asymmetric key cryptography. The device that sends the key uses the route `/get/account/<user_id>/key/send` in `crypto_server` to connect with the server, and the receiving device uses the route `/get/account/<user_id>/key/get` in its own copy of `crypto_server` to also connect with the server. The PreVeil server at `wss:// collections . preveil .com/relay_ws` is used as a relay websocket between the browser websockets on the two communicating devices. We cannot guarantee that secure WebSockets are secure against all man-in-the-middle attacks (MITM server can masquerade as PreVeil server, or untrustworthy Certificate Authority is used).

PreVeil uses additional steps to protect the private key during the message passing between the old device and the new device. This code is contained in the `sendSecureMessage()` function in `send_keys_handler`, which is called by both `sendKeys()` used for key transfer and `sendShard()` used for approval groups in `crypto_server`, so this process is similar to the one detailed in Section 4.5. The receiver simultaneously calls the corresponding function `getSecureMessage()` in `get_keys_handler`. After both devices are connected via the websockets, they each create anonymous one-time use private/public key pairs. The sender and receiver wait for the each other's public key hash as a pre-handshake, and then they send their actual public keys, which are verified against the hashes to complete the handshake. Then, to verify that the correct devices are connecting, a pin verification step occurs (the pin is like a shared secret key). The receiver sends a pin to the websocket. The sender listens for the pin, and if the pins match, it encrypts and sends the secret message to the websocket. The receiver waits for this encrypted message, decrypts it when received, and stores the private key on the device.

Since PreVeil promises end-to-end encryption, we must ensure that the PreVeil servers have no way of receiving the user's private key at any time, even if for a moment. The encrypted message that is passed through the Websockets connection contains the encrypted private key. In the function `sendKeys()` in `crypto_server`, the private key is retrieved from the device's local storage and then directly passed to the `sendSecureMessage()` protocol described above. Since `sendSecureMessage()` operates on the device rather than on the server, the private key never leaves the device unencrypted. Since the PreVeil server is used as a relay between the two communicating devices, PreVeil may be able to see the encrypted private keys. Although this may be potentially risky, we don't think it is a serious risk, since each private key that is sent in a message is encrypted with a one-time use key for that specific message. It is unlikely that an adversary would be able to guess the decryption function for each encrypted private key.

4.7 Secure Updates

PreVeil's update installer should not provide attackers opportunities to introduce malicious updates. In order to install updates securely, an authenticated connection and/or authenticated downloads are recommended to ensure integrity and authenticity.

The updater server at `https://deploy. preveil .com:8080` uses HTTPS with SSL to connect with the user's browser, meaning that the connection is authenticated. The downloads themselves are

also authenticated using asymmetric key cryptography. Each update sent by the server is signed by PreVeil's private key, and the user's device verifies the signature using PreVeil's public key.

PreVeil's public key is stored on the user device at in a file called `verifier_prod.txt` in the `updater/conf` directory. This file is placed during the initial installation, preventing a man-in-the-middle attacker from masquerading as PreVeil in future updates. Unless this public key is somehow manipulated, we can ensure that it is indeed PreVeil who is managing the servers providing the downloads.

The code which fetches updates from the server is in `remoteFetcher`. At a high level, first, `getUpdateMeta()` allows the updater to determine if the version being served is any good. Then, the updater has the option of pulling the download using `getUpdate()`. After the download is pulled, then the signature is verified, and only then is the update actually installed. If the signature is invalid, then the download is not installed, so that when a user runs PreVeil, none of the code in the invalid download is run.

In more detail, the method that starts the updater server in `remoteFetcher` first checks that the user is running as root. It finds the signature verifier key path, uses libnacl to load a key from the path, and starts an update loop that calls `getUpdateMeta()` and `doUpdate()`. The `getUpdateMeta()` method pulls from the URL `https://deploy.preveil.com:8080/getUpdateMeta` using the current version as a parameter. Then, it checks the status: "error", "ok" (up to date), or "update" (update available). If an update is available, it checks the version and signature of the update, and returns metadata about the update. Then, the `doUpdate()` method uses the update version to download data from `https://deploy.preveil.com:8080/getUpdate`. The `doUpdate()` method also takes the signature verifier key as its input. It verifies the signature contained within the update metadata, which is packaged along with the update data. The update is started only when the signature is verified. The method `applyUpdate()` installs the verified download.

We noticed in the comments of the code in `remoteFetcher` that it was noted that a failure mode occurs when $client\_count == max\_clients$ on the server, for a version that could upgrade the client. Apparently, the server then returns an error, and the client will decide not to proceed with the download, even if an upgrade is available. If this bug has not been fixed yet, it could be a potential risk, because it is crucial that users are able to upgrade their software at all times. We can imagine a scenario where an important software patch is released, fixing a vulnerability, and the user cannot pull the update.

4.8 Frontend

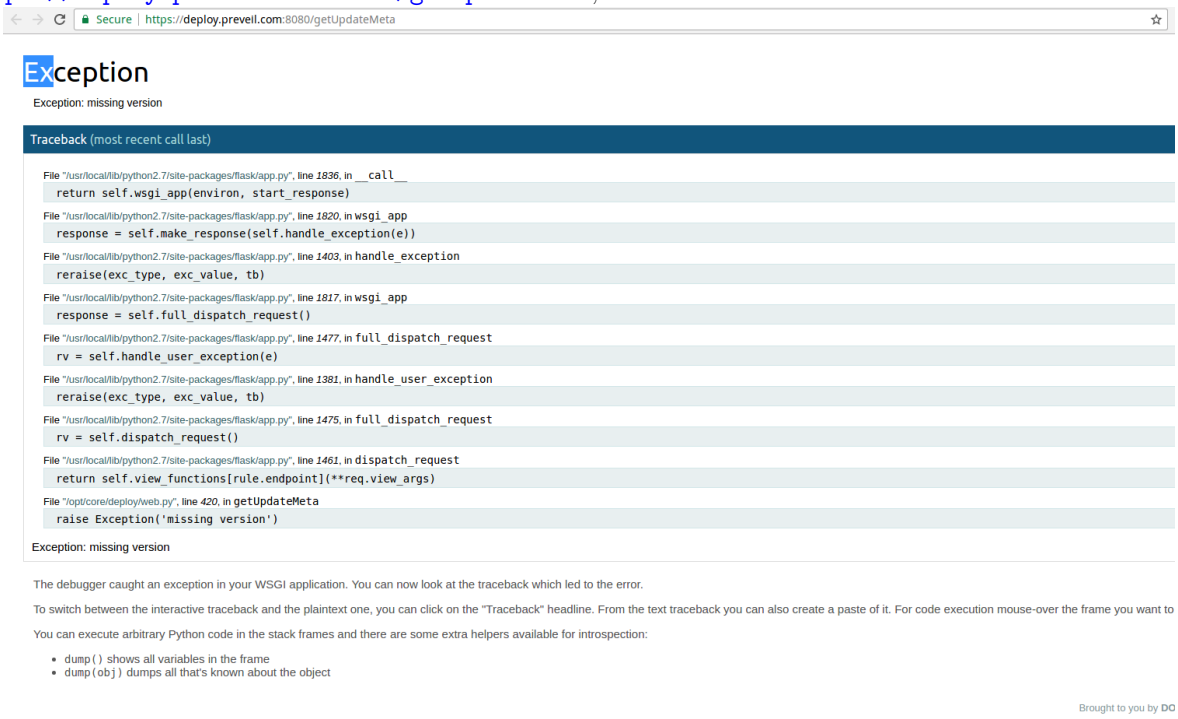
The primary danger in the front-end is the injection of arbitrary JavaScript code, which might send sensitive information to an attacker's server or make a local request to the `crypto_server` that sends an email on your behalf. This can occur if, for example, someone sends you an email containing script tags and your email client doesn't properly them sanitize them. We tested the PreVeil client for injection vulnerabilities by sending several emails containing script tags in various places (the email body, subject, attachment names, etc.) and couldn't break anything. We found that the email body is rendered inside an `iframe`, which completely sandboxes any JavaScript that it might contain, and that all the other email fields are properly sanitized.

5 Results

We extensively explored several parts of the codebase and couldn't find any major security flaws. We did, however, document what we looked at and why we deemed it secure, and hope that the PreVeil team will benefit from those assurances. We did find a minor security problem outside of the client, which we document below. We also briefly discuss a simple test we applied to ensure that all outbound emails got properly encrypted.

5.1 Debugger left running in production

We found a Werkzeug debugger running in production on https://deploy.preveil.com, which was revealed when we triggered a minor crash on the server. This has the potential to be a major security hole because the debugger allows for arbitrary remote code execution, but the PreVeil developers are using an up-to-date version of WerkZeug which password-protects the debugger with a PIN that's printed in the server-side logs (we looked into potentially cracking the PIN, but it's nine characters long and Werkzeug limits you to only 10 incorrect authentication attempts). The target URL was https://deploy.preveil.com:8080/getUpdateMeta, and below is a screenshot:



5.2 Outbound email test

We conducted a simple test by modifying the PreVeil client to save all outbound emails to disk so that we could manually look them over. We added a few lines of code to `util.PreparedMessage.upload()`, which is responsible for uploading emails to the PreVeil server, that saved each data block of the email before it got sent out. We then sent a few emails in the client, and checked the saved versions to make sure that everything inside was encrypted (and that no parts of the email were accidentally sent out as plaintext).

6 Conclusion

In our security review, we found no vulnerabilities in the PreVeil client, under the assumption that the underlying primitives, such as libnacl and SSL, are secure. We did find that PreVeil's online description for key transfer and approval groups differed somewhat from how they were actually implemented. We also stumbled upon an online debugger while looking through the way that updating works, which is a potential point of attack on the server. We will be looking for future developments from Preveil with interest.