An End-to-End Encryption Scheme for SecureDrop

Andrés Náter　　Erica Santana　　Patrick Wahl

May 2018

Abstract

In an age of heightened surveillance and a questionable right to privacy, whistleblowers like Chelsea Manning and Edward Snowden become important figures in the fight for government transparency and accountability. However, many face fears of being identified and silenced before being able to tell their stories. SecureDrop is a platform allowing whistleblowers to communicated with journalists anonymously and without fear of retribution from higher powers. However, SecureDrop is not end-to-end encrypted, that is, messages and files sent from the whistleblower to the journalist are plaintext and can be read easily if an adversary can compromise the server before it encrypts them. Although SecureDrop's primary concern is with preserving *anonymity* of the user rather than hiding the materials, identification of the messages could lead to efforts to shut down a story before the public is able to hear about it. We therefore propose and prototype a secure way of sending encrypted messages to journalists using the SecureDrop platform. We develop a browser extension which, when used in conjunction with SecureDrop, can help prevent sources from being deanonymized through Tor browser JavaScript watering hole attacks. We evaluate the tradeoffs in security and ease-of-use of SecureDrop under our new scheme.

1 Introduction

Since Daniel Ellsberg famously released the Pentagon Papers in 1971, whistleblowers, or those who release secret information to the public (often knowing they face severe punishment), have become essential in the fight for holding those in power accountable for their actions. This has led to organizations cracking down on those who intend to leak classified information, and thus to new ways of leaking said information anonymously and securely.

2 SecureDrop

2.1 Background

SecureDrop is an open-source platform from the Freedom of the Press Foundation. It is designed for whistleblowers, from here on referred to as “sources,” to share sensitive information with journalists and media organizations. The platform aims to provide safer journalist-source communication by reducing the metadata trail and eliminating third parties. SecureDrop’s workflow includes six separate parts, as seen in Figure 1, but we will focus primarily on two of them: the source area, where sources access the web page and upload documents, and the SecureDrop area, containing the server where files are encrypted and stored.

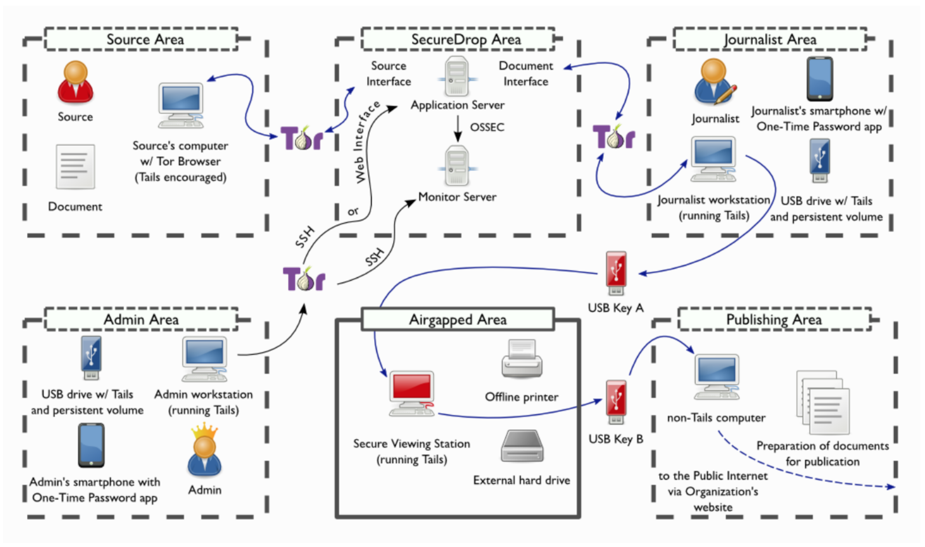

Source Area
Source
Source's computer w/ Tor Browser (Tails encouraged)
Document
SecureDrop Area
Source Interface
Document Interface
Application Server
OSSEC
Monitor Server
Web Interface
or
SSH
SSH
Journalist Area
Journalist
Journalist's smartphone w/ One-Time Password app
Journalist workstation (running Tails)
USB drive w/ Tails and persistent volume
USB Key A
Admin Area
USB drive w/ Tails and persistent volume
Admin workstation (running Tails)
Admin's smartphone with One-Time Password app
Admin
Airgapped Area
Offline printer
Secure Viewing Station (running Tails)
External hard drive
USB Key B
Publishing Area
non-Tails computer
Preparation of documents for publication
to the Public Internet via Organization's website

Figure 1: SecureDrop Architecture

SecureDrop requires all communication over the internet to be routed through the Tor Browser. Tor Browser is based on Mozilla Firefox, and routes all traffic through the Tor network, which encrypts the data, source, and destination addresses multiple times and sends them through a series of relays which decrypt one layer of data at a time.

In this way, none of the nodes in the network can figure out both the source and destination addresses, and neither can those surveilling the data being sent, allowing SecureDrop users to remain anonymous.

Journalistic organizations, such as The New York Times, can set up SecureDrop instances for use by sources wishing to contact them. The platform uses two physical servers, one public-facing server that stores messages and documents and another one which monitors the security of the public-facing one.

The public-facing server is a web application and Tor Hidden Service that accepts messages and documents from sources who use Tor. The server encrypts the data received using GPG, which is an implementation of the OpenPGP standard. The use of Tor protects the user's data from from third-party attackers in transit.

As illustrated in Figure 2, the source submits files through Tor, which are then sent to the news-outlet server through the internet. Once they reach the server, they are encrypted with the journalist's public key and stored on the server. The journalist can then access the encrypted files and retrieve them using a flash drive. To view the files, the journalist makes use of a computer that is air-gapped, i.e. not connected to the internet, the only computer which contains the private key and is therefore able to decrypt the files.

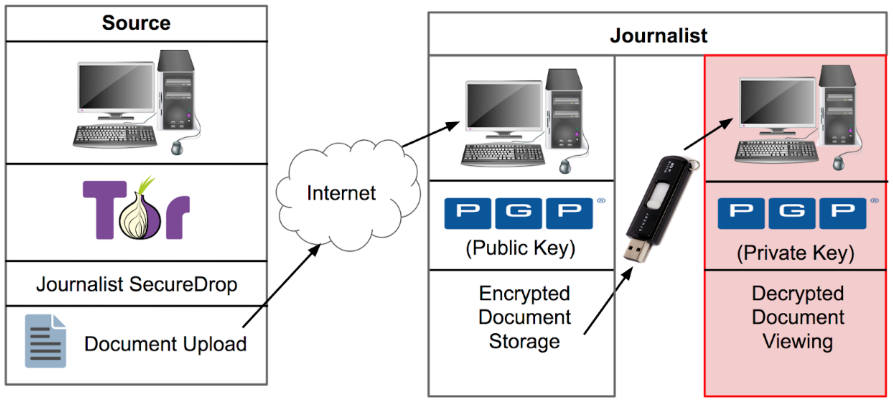


Figure 2: A closer look at the architecture we are concerned with

Sources are given a codename which allows them to establish a connection with a particular news organization. That same codename can be used afterwards by the source to read any messages from the journalists or to send more information. All the documents, messages, and responses from a codename are grouped together into a collection. It is important to note that the journalists have a distinct codename to identify the source.

2.2 The need for end-to-end encryption

SecureDrop is currently not end-to-end encrypted. The files and messages whistleblowers submit are only encrypted once they reach the news outlet server. This potentially enables an adversary to intercept the files and messages when they reach the server but before they are decrypted, thus gaining access to sensitive information. One way to prevent this is to encrypt files and messages before they are sent to the journalist, on the client side. This would be an end-to-end (E2E) encryption scheme.

One method of providing E2E encryption outside the browser would be for

SecureDrop to provide a client application which provides an anonymous connection to the server and client-side encryption functionality. This would be very secure, compartmentalized, and easy to use, even for the non tech-savvy user. However, there is a serious problem. If a government entity were to discover that a SecureDrop user has an application installed on their machine to be used for the express purpose of leaking secret information, the entity could use this information to incriminate the source.

A safe way sources can currently achieve E2E encryption is by encrypting files using PGP on their own computer, then uploading them the normal way. SecureDrop provides a public key allowing users to do this. The security here would be airtight — even if an adversary were to intercept the message, they would be unable to decrypt it without knowing the private key (which is stored on the airgapped computer and thus extremely difficult to access). The problem is that this might seem threatening to users with no experience with cryptography or computing.

For this reason, the SecureDrop team has suggested providing an E2E encryption scheme that runs in the source's browser. However, this poses a security risk: client-side encryption via a web browser requires JavaScript to be enabled in the source's browser. JavaScript has been used previously in watering hole attacks designed to deanonymize Tor users [1], so SecureDrop recommends that the source disables JavaScript due to the risk of these attacks.

If the user could be absolutely sure that the JavaScript they are running is signed by the developer (SecureDrop), then they could securely run the code without a chance of being identified by adversaries. We worked on making SecureDrop E2E encrypted by creating a way for users to verify that the JavaScript their browser is running is signed by the developer, and then implementing this signing logic in SecureDrop's software.

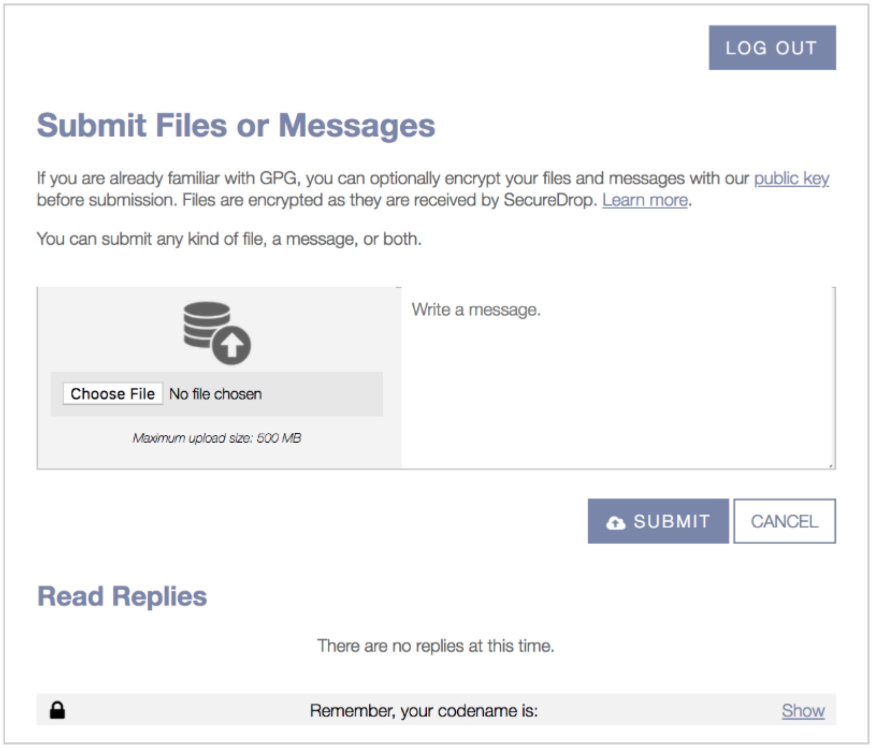
3 Implementation

Our implementation consisted of three parts:

1. Client-side logic to encrypt data on the source's computer
2. Server-side logic to sign pages and send the signature to the source
3. Browser extension to verify signatures based on page content

3.1 Client-side encryption

In the current platform, sources simply upload a file and submit it to the journalists. However, sources are already provided with the journalist's public key. They are encouraged to download this public key and use GPG encryption software to encrypt their files on their own machine, before submitting. Implementing client-side encryption thus consists of three main steps: giving the source the option to have encryption done before submitting, retrieving the journalist's public key, and actually encrypting the file. Note, however, that before the source is able to make any choice about encryption on their side, they must enable JavaScript. They are made aware that enabling JavaScript may come with risks and are provided instructions on how to check the authenticity of the submission page before enabling JavaScript.



Submit Files or Messages

If you are already familiar with GPG, you can optionally encrypt your files and messages with our public key before submission. Files are encrypted as they are received by SecureDrop. Learn more.

You can submit any kind of file, a message, or both.

Read Replies

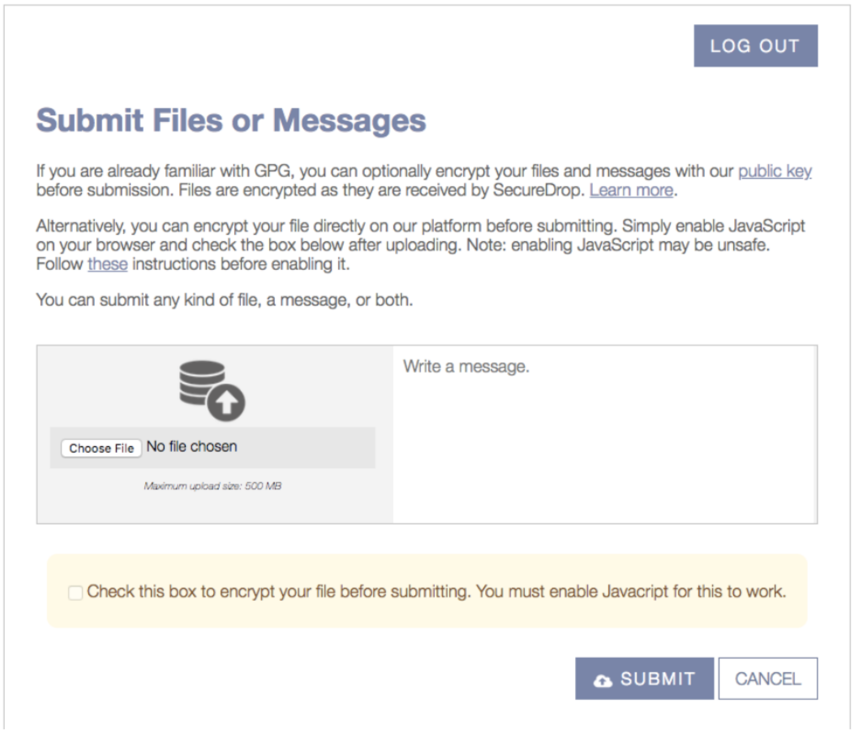
There are no replies at this time.

Remember, your codename is:

Figure 3: The original document upload interface

As shown in Figure 4, a checkbox was added in the source submission page to let the source decide whether to encrypt or not. This checkbox lives in a yellow box to emphasize that checking it is an active decision and not a requirement or consent. Screens previous to the "lookup" page make use of this yellow box to emphasize important messages to the source.

Once a file is uploaded, if the page has been verified and JavaScript has been enabled, the checkbox becomes available and the source can choose to encrypt or not. If the source chooses to encrypt, the process is executed in the background without the source having to go through any additional steps. First, the journalist's public key is retrieved. The key is retrieved by employing a XMLHttpRequest object, used in web development to interact with servers and retrieve data. In this case, it helps retrieve the journalist's public key from the URL that contains it. The journalist's public key is an ASCII-armored PGP



Submit Files or Messages

If you are already familiar with GPG, you can optionally encrypt your files and messages with our public key before submission. Files are encrypted as they are received by SecureDrop. Learn more.

Alternatively, you can encrypt your file directly on our platform before submitting. Simply enable JavaScript on your browser and check the box below after uploading. Note: enabling JavaScript may be unsafe. Follow these instructions before enabling it.

You can submit any kind of file, a message, or both.

☐ Check this box to encrypt your file before submitting. You must enable Javacript for this to work.

Figure 4: Updated interface with checkbox to turn on client-side encryptions and link to verification instructions

key of the form:

```
   -----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFJZi2ABEACZJJA53+pEAdkZyD99nxB995ZVTBw60SQ/6E/gws4kInv+YS7t
wSMXGa5bR4SD9voWxzLgyulqbM93jUFKn5GcsSh2O/lxAvEDKsPmXCRP1eBg3pjU
+8DRLm0TEFiywC+w6HF4PsOh+JlBWafUfL3vwrGKTXvrlKBsosvDmoogLjkMWomM
KBF/97OKyQiMQf1BDJqZ88nScJEqwo0xz0PfcB04GAtfR7N6Qa8HpFc0VDQcILFB
0aJx5+p7nw1LyR37LLoK8JbEY6QZd277Y0/U+O4v6WfH/2H5kQ8sC+P8hPwr3rSg
.   .   .
n1xbbX4GXQl3+ru4zT6/F7CxZErjLb+evShyf4itM+5AdbKRiRzoraqKblBa4TfJ
BSqHisdcxdZeBe19+jyY6a8ZMcGhrQeksiKxTRh7ylAk7CLVgLEIHLxXzHoZ0oAF
z2ulG+zH9KS9Pe8MQxHCrlyfoQElQuJoYbrYBOu28itvGPgz6+5xgvZROvPoqIkI
k8DYt9lJqUFBeZuFJd5W1TuHKLxueVYvSKeG+e3TjOYdJFvDZInM4cNWr8N92mYS
iphljiHAKVTQeIf1ma07QUH/ul3YC+g07F+BLonIIXA6uQVebv5iLxTg0zIQwHTJ
Vu4MPiQNn1h4dk1RonfV/aJ+de1+qjA8
=XVz8
-----END PGP PUBLIC KEY BLOCK-----
```

After retrieving the journalist's public key, the input from the file submission section is fetched and converted into Uint8Array format so that it can be encrypted. Encryption is achieved using the OpenPGP.js library, an open-source JavaScript implementation of the PGP standard, employed in the SecureDrop project. The library is included in the ¡script¿ portion of the "lookup" page, where the source uploads the file. Once the source file is encrypted, it is converted into a file to keep consistency with submission protocols. With the creation of this file, the client-side encryption process is completed.

In order for this feature to be deployed, the official submission protocol needed to be modified. Currently, the backend retrieves the original file uploaded. The file to be retrieved will depend on the value of the checkbox: checked or not. If the value is checked, the encrypted file will need to be retrieved, otherwise the original file can be retrieved. Once the correct file is retrieved, it can be sent to the server for storage and for the journalist to obtain.

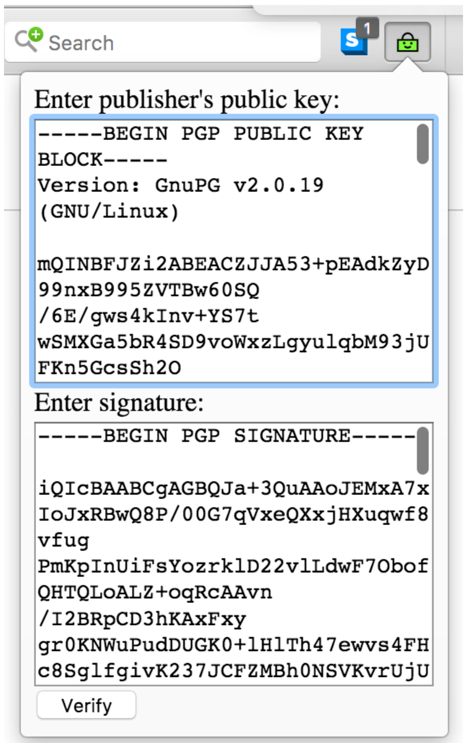
3.2 Server-side page signing

Page signing is very simple. The server generates the lookup page as it would if the user accessed that page. Instead of sending it as a response, however, it searches through the page for `<script>` tags, combines what is inside these scripts into a single string, and then signs that string using the GPG module SecureDrop uses for other cryptographic purposes. One problem we encountered is that signing is done with a private key, which SecureDrop stores on an air-gapped computer, rendering the server unable to access it for signing. For document signing, then, we must create a new key pair for each user, using the private key to sign and remembering the user's session ID in order to send them the corresponding public key for verification. The signature is detached and both the public key and the signature are ASCII-armored for ease of integration with third-party verification extensions.

3.3 Verification browser extension

We created an example prototype of a third-party Firefox/Tor browser extension that could be used to verify that the page will run only JavaScript that is signed by the journalist's SecureDrop server. Called SafePage, its user interface is simple: it takes as input a PGP public key and signature, both ASCII-armored, and prints whether or not the page's inline JavaScript is verified as genuine. If the JavaScript is genuine, the user can turn on JavaScript for the page to perform client-side encryption. The extension uses the OpenPGP.js library, an open-source JavaScript implementation of the PGP standard, for verification.

It works by reading in the ASCII-armored public key and signature the user enters, and then injecting a content script into the web page which grants it access to the document object model (DOM). We had originally planned to sign and verify the entire web page document, but it is unfortunately not possible to access the source code file as sent by the server. Instead, content scripts only



Enter publisher's public key:

```
-----BEGIN PGP PUBLIC KEY
BLOCK-----
Version: GnuPG v2.0.19
(GNU/Linux)

mQINBFJZi2ABEACZJJA53+pEAdkZyD
99nxB995ZVTBw60SQ
/6E/gws4kInv+YS7t
wSMXGa5bR4SD9voWxzLgyulqbM93jU
FKn5GcsSh2O
```

Enter signature:

```
-----BEGIN PGP SIGNATURE-----

iQIcBAABCgAGBQJa+3QuAAoJEMxA7x
IoJxRBwQ8P/00G7qVxeQXxjHXuqwf8
vfug
PmKpInUiFsYozrklD22vlLdwF7Obof
QHTQLoALZ+oqRcAAvn
/I2BRpCD3hKAxFxy
gr0KNWuPudDUGK0+lHlTh47ewvs4FH
c8SglfgivK237JCFZMBh0NSVKvrUjU
```

Figure 5: The SafePage prototype's interface

have access to the DOM, a browser-parsed version of the code inside the `<html>` tags. While this presents a security risk where a malicious entity could inject JavaScript outside the `<html>` tags, the risk is mostly or entirely mitigated by the fact that modern browsers (including Tor) will place scripts outside the `<html>` tags into the body of the document, inside the DOM. There is also the issue of the `<!DOCTYPE>` tag, which goes outside the `<html>` tags. We tried using JavaScript's XML Serializer to read the whole web page document, but this causes alterations in the source HTML which would prevent valid signatures from verifying.

Finally, we decided upon signing only portions of the document inside of `<script>` tags. This content is not altered when the browser parses the document to create the DOM, and so will be the same both when it is signed and when it is verified. However, there is a catch: if we only look at the content inside the script tags, that leaves the page open to vulnerabilities if the tag loads a malicious source inside itself, e.g. `<script src='www.evil.com/bighack.js'>`. Therefore, we must sign and verify both the tag contents and the tags themselves.

The browser extension needs to be third-party, rather than SecureDropsupported, for precisely the same reason SecureDrop cannot just make a non-browser client application to upload encrypted documents: so that the user will not have any incriminating software downloaded on their computer. With a third-party application, even one designed to be used with SecureDrop, there is plausible deniability that its intent is to circumvent legal boundaries — it could have been installed to be used for other purposes; verifying web pages' JavaScript is a good security practice even for law-abiding citizens!

4 Argument for Security

On the continuum of possible encryption schemes for SecureDrop, we believe our implementation is more secure but less user-friendly than no E2E encryption at all, but less secure and more user friendly than manual PGP encryption before uploading.

An adversary cannot inject malicious JavaScript into the page without altering the page's signature and preventing it from signing properly, since the verification extension reads every script tag and uses their contents in verifying. There is a chance that an adversary could send a malicious public key and signature along with malicious code. To solve this problem, SecureDrop could transmit its public verification key from a secure, centralized server, which would be much more difficult to compromise. This vulnerability is also somewhat present in the current SecureDrop implementation, which sends the public key for manual PGP encryption. An adversary could send a malicious public key, intercept the encrypted file, and use their private key to decrypt the contents. This is, however, outside the scope of the problem we are trying to solve.

Otherwise, security is in the hands of the user — if they do not enable JavaScript unless they have verified the page's signature, then there is little chance that they will run malicious code in their browser.

5 Discussion/Conclusion

SecureDrop is an open-source platform. There is a community of coders and contributors who discuss improvements and looks for bugs regularly. As we evaluated SecureDrop and a potential project based on the platform, we found that end-to-end encryption has been discussed before. Many were against it because of the risks posed by the use of JavaScript and possible exploits with the goal of deanonymizing sources. Identification of sources in such a context could lead to dangerous situations for these sources, including life-threatening ones. The stakes are quite high when it comes to whistleblowing. However, we also believe that end-to-end encryption is essential for more robust security. Furthermore, a crucial part of having a platform like SecureDrop be effective is having it be as user-friendly as possible. A source who is not tech savvy might have some trouble downloading a key and encrypting their own files on their machine. Thus, the only way to achieve end-to-end encryption, while removing the burden of encrypting their own files from the source, seemed to be to implement client-side encryption in the source app using JavaScript. As long as the necessary precautions, this seems like a sensible and feasible way to achieve end-to-end encryption for SecureDrop.

References

[1] Dan Goodin. Attackers wield firefox exploit to uncloak anonymous tor users, Aug 2013.