Pentest-Report SecureDrop 12.2013

Cure53, Dr.-Ing. Mario Heiderich / Nikolai K. / Fabian Fäßler

Index

Intro

“SecureDrop is an open-source whistle-blower submission system managed by Freedom of the Press Foundation that media organizations use to securely accept documents from anonymous sources. It was originally coded by the late Aaron Swartz.”
From https://github.com/freedomofpress/securedrop

This test was carried out by three Cure53 testers and took place over a 5-day period between 2nd and 6th of December 2013.

The test focused on application and server security matters directly related to the code and features of the SecureDrop application. The Cure53 team was granted an access to an existing system setup, including a submission form, a document interface and console access to the App- and the Monitor-Server[1]. The tests were performed from both black-box and white-box perspectives. While testing the application, the Cure53 team had consulted a set of sources used for the tested setup. Furthermore, our team was able to navigate the servers’ directories via SSH access with a high-privilege user

[1] SecureDrop Install. Guide https://github.com/freedomofpress/securedrop/blob/master/docs/install.md

account. The Cure53 team further followed the full installation and setup manual in order to live-experience all components that are crucial to the SecureDrop communication system. Those included the use and analysis of Tails USB sticks and a server setup on a set of AWS instances.

This report does not mention vulnerabilities already reported in “DeadDrop/StrongBox Security Assessment” by Czeskis et al., regardless of their fix status[2]. We similarly do not elaborate on the SecureDrop documentation, installation process, code management or hypothetical technical proficiency of SecureDrop users, such as the perceived degree of concurrence between the complexity of the installation and the technical acumen and background of an average user. We embrace a similar threat model to that outlined by Czeskis and agree on the assumptions made about the attacker’s strengths, competencies and exceptionally high motivation. However, we equally consider a scenario in which a journalist goes rogue or is bribed by a third party - the potential cases that position him or her as inclined to unveil secrets behind a source. This inevitably led to a discussion about possibly infected PGP keys briefly mentioned in the conclusion of this Report.

The penetration-test yielded an outcome of eleven weaknesses. Significantly, none of them are considered critical. The SecureDrop team managed to fix several of the reported issues during the penetration-test already, thus allowing for a timely verification from the Cure53 team’s side.

Scope

- **SecureDrop Application**
  - https://github.com/freedomofpress/securedrop
- **Source Interface**
  - http://jgdevcded4x6dl37.onion
- **Document Interface**
  - http://ugdhee7iyiiokyty.onion:8080
- **Admin app server**
  - http://vhvxgl6apivhewlx.onion
- **Admin monitor server**
  - http://4mgzyj4p6mcl67qt.onion

[2] DeadDrop/StrongBox Security Assessment http://homes.cs.washington.edu/~aczeskis/research/pubs/UW-CSE-13-08-02.PDF

Test Chronicle

- 2013/12/02 - Penetration-Test starts
- 2013/12/02 - Source code audit against Python files begins
- 2013/12/02 - Checking for safe random number generation
- 2013/12/02 - Auditing Tails Browser Configuration for leaks
- 2013/12/02 - Checking HTTP Security headers
- 2013/12/02 - Checking for insecure defaults in install files
- 2013/12/02 - Checking for unnecessary privilege spills
- 2013/12/02 - Analyzing transport security issues
- 2013/12/02 - Checking file permissions on default install
- 2013/12/02 - Checking general app-server setup/perms and chroot configuration
- 2013/12/02 - Testing session security and file upload security
- 2013/12/02 - Searching RCE bugs via Flask
- 2013/12/02 - Seeking XSS and SQLI Bugs via File Upload
- 2013/12/03 - Checking OSSEC configuration on app-server
- 2013/12/03 - Quick audit of the OSSEC sourcecode
- 2013/12/02 - Checking general monitor-server setup and permissions
- 2013/12/03 - Verifying whether WFP in Tor applies to SecureDrop
- 2013/12/03 - Analyzing 404 headers for possible misconfiguration
- 2013/12/03 - Analyzing a recurring 500 error during a submission check
- 2013/12/03 - Referer leakage analysis, link injections, link hijacking
- 2013/12/03 - More tests against possible LFI / directory traversal via verify()
- 2013/12/03 - Tests against freshly released Tails 0.2.2 OS version
- 2013/12/03 - Further research on possible traffic analysis / confirmation attacks
- 2013/12/04 - Tests against command injection via OpenPGP
- 2013/12/04 - Studies on possibly poisoned PGP keys / chameleon-keys
- 2013/12/04 - Tests against possible python routing bugs
- 2013/12/04 - Tests with broken UTF-8 / Unicode
- 2013/12/04 - Tests of the recently fixed messaging feature
- 2013/12/04 - Ongoing source code audit
- 2013/12/04 - Tests against possible path-traversal in ZIP files
- 2013/12/05 - Ongoing monitor-server checks
- 2013/12/05 - Reviewing *iptables* configuration
- 2013/12/05 - Ongoing source-code audit
- 2013/12/05 - Reviewing dependencies
- 2013/12/05 - Reviewing the Flask session backend
- 2013/12/06 - Tests against cookie security and possible leaks
- 2013/12/06 - Final source-code audit
- 2013/12/06 - Checks against recent commits
- 2013/12/06 - Finalization of the Pentest-Report
- 2013/12/06 - End of the Penetration-Test

Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact, which is simply given in brackets following the title heading for each vulnerability. Each listed vulnerability is given a unique identifier for the purpose of facilitating follow-up correspondence.

SD-01-001 No HTTP Security Headers on Apache Error Pages *(Medium)*

The SecureDrop submission interface uses several HTTP Security headers to ensure additional level of protection against Clickjacking, XSS and similar attacks. However, upon submitting a faulty request, an Apache default error page will be delivered.

Application Headers:

```
HTTP/1.1 200 OK
Date: Mon, 02 Dec 2013 12:08:45 GMT
Server: Apache
Expires: -1
Pragma: no-cache
Cache-Control: no-cache, no-store, must-revalidate, max-age=0, no-cache, no-store, must-revalidate
Set-Cookie: session=.eJw9zE0LgjAAgOG […] OdyUvKGRk; HttpOnly; Path=/
x-frame-options: DENY
Access-Control-Allow-Origin: http://jgdevcded4x6dl37.onion:80 < redundant
X-XSS-Protection: 1; mode=block
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 1090
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

Error-Page Headers:

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 02 Dec 2013 12:02:23 GMT
Server: Apache
x-frame-options: DENY
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 343
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Tampering with the CSRF token or changing other parameters to invalid values can, among other instances, provoke such behavior. By default the Apache does not use any HTTP security headers except for the X-Frame-Options. It should be endured that all server errors are handled by custom error pages, as otherwise an attacker can abuse the lack of headers to lever an attack.

This problem relates to the following SD-01-002 vulnerability and issue #107:
https://github.com/freedomofpress/securedrop/issues/107

SD-01-002 Missing HTTP Security Headers and Name-Randomization *(Low)*

Tracked in issue #107 on GitHub

In addition to the HTTP Security Headers that are in use already, we recommend to set yet another pair of headers enhancing the client-side security of the application users. Those include the specific:

- *X-Content-Type-Options: nosniff*[3]
- *X-Download-Options: noopen*[4]

A similar issue pertains to *window.name* variable not being presently randomized. This potentially is a door-opening for an attacker who seeks to utilize TabNabbing attacks[5]. As it stands per current recommendation, the application is to be run without any JavaScript switched on. However, in the case of TabNabbing attacks, this may aid the attacker who benefits from the impossibility of client-side mitigation mechanisms being in place.

Other attacks, for instance the referrer leakage via HTML link injection/image injection (see SD-01-008), similarly work without JavaScript activated. As such, for instance CSS injection allows for severe data leakage occurrence if the attacker manages to inject complex CSS and style directives.

Randomizing window.name:

```
<script type="text/javascript">
       window.name = '%unique_random_value%';
</script>
```

Rather than urging users to switch off JavaScript, it is recommended to consider using CSP headers instead. JavaScript execution in itself is not considerably problematic, that is if there is no exfiltration channel for potentially leaked client-side data (XSS, cookies, local IP via WebRTC etc.). CSP takes care of that very problem by forbidding injected, non-same domain JavaScript and prohibiting cross-domain data leaks.

This problem is closely related to the SD-01-001 vulnerability and has already been mentioned in issue #107: https://github.com/freedomofpress/securedrop/issues/107

[3] MIME Sniffing Risks: http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx

[4] IEBlog: http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx

[5] TabNabbing http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/

SD-01-005 Missing HTTP Security Headers for 404 Pages *(Medium)*

Reflecting on the issues described in SD-01-001, one notes an important similarity for that Apache error pages are not the only ones not being applied with HTTP security headers. Same holds true for the 404 pages available in both the Source- and the document interface cone almost entirely void of the required header setup. The Apache default does not use any HTTP security headers apart from the X-Frame-Options.

Example URLs:

- http://ugdhee7iyiiokyty.onion:8080/HELLO
- http://jgdevcded4x6dl37.onion/static/

It is a must that each possible server status that provokes default page rendering is being handled by a custom page or, alternatively is supplied with the additional headers. This issue was spotted after SD-01-001 was addressed and fixed.

Continuing this argument, it is notable that the application sets several security- and privacy-relevant headers via Python:

```
@app.after_request
def no_cache(response):
        """Minimize potential traces of site access by telling the browser not to
        cache anything"""
        no_cache_headers = {
        'Cache-Control': 'no-cache, no-store, must-revalidate',
        'Pragma': 'no-cache',
        'Expires': '-1',
        }
        for header, header_value in no_cache_headers.iteritems():
        response.headers.add(header, header_value)
        return response
```

This should be avoided and, instead, the headers should all be set at a central position by the web-server, ensuring a single rather than many locations for maintaining headers and ascertaining that the same headers are not set multiple times. The latter is crucial due to its potential for invalidation and making headers' and their effect void in several browsers.

SD-01-006 Possible path confusion/traversal via imprecise *store.verify()* *(Medium)*

Tracked as issue #194 on GitHub

The method *store.verify()* checks file paths provided via URL and other ways, raising an exception if they cannot be matched against the validation criteria.

A problem with this validation process was spotted: *os.path.commonprefix()* is not sufficient for checking if the path is inside the configured store path. By performing comparisons on a 'character by character' basis only, it allows navigation into another folder whenever they share the same start string.

Example: `config.STORE_DIR = '/opt/store'`
PoC: `store.verify('/opt/store_backup')`

A working mitigation mechanism has to guarantee that the path points to a location inside the configured store folder. A possible way to resolve this would be to add another check in *store.verify()* with *os.path.relpath(p, config.STORE_DIR).* If the absolute path is not inside the store directory, os.path.*relpath()* will return a string starting with '../'.

Example:
os.path.relpath('/opt/store_backup', *config.STORE_DIR*) ==
'../store_backup'

SD-01-008 HTML Links on SecureDrop static sites leak Referrer *(Medium)*

Tracked as issue #195 on GitHub

The SecureDrop static pages contain links pointing to external HTTP websites. This can be considered an information leak, essentially assisting the user de-anonymization process via the HTTP referrer and the DOM *document.referer* property.

Examples:

- http://jgdevcded4x6dl37.onion/tor2web-warning
- http://jgdevcded4x6dl37.onion/howto-disable-js

Clicking such link will open the tor2web website in the very same browser window. Consequently, a leak of the referrer to this website will take place, allowing those third-parties to have control over the log files and website markup. This can be equated to knowing that the user visited the TOR hidden service right before visiting a given website. Another link is pointing to the website of the TOP project - again as plain HTTP URL:

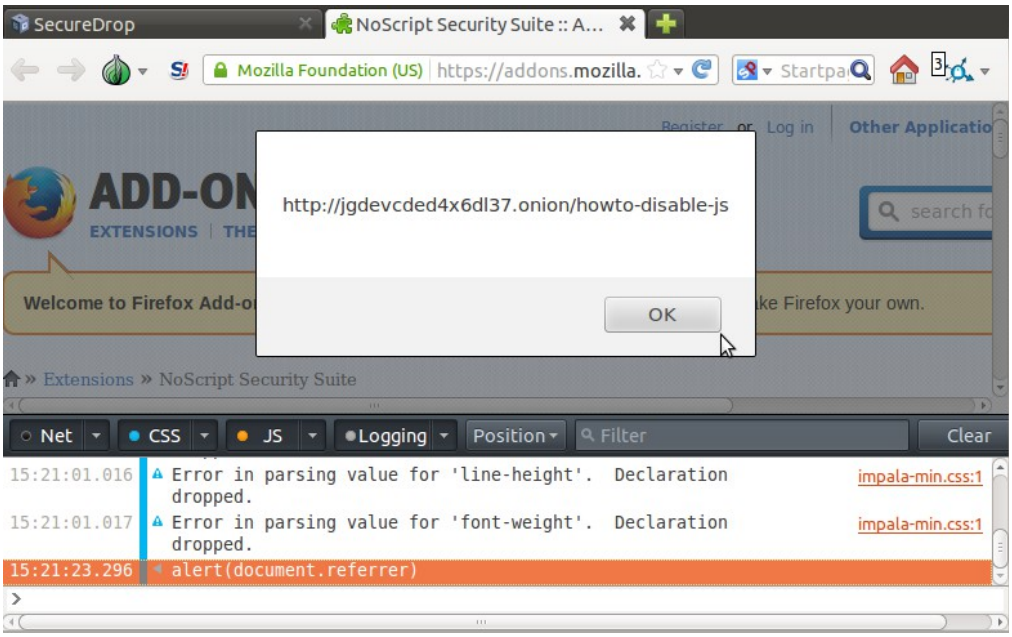

Fig.: A website linked on SecureDrop is now aware of where the user has come from

Using HTML links or any other external resources or pointers in the markup of the SecureDrop application should be eliminated at all cost. Users should be encouraged to simply copy and paste a HTTP URL shown in plain-text. It could also be considered to employ Data URIs to create a referrer-less link and avoid the data leakage to the outside world this way.

Note that the HTML specification also mentions the “noreferrer” attribute for links[6]. This might be then used to attempt provision of non-leaky links, although proceeding with caution is advised here for not all modern browsers support this attribute fully as of yet. Therefore, relying solely on this attribute to work as expected is not recommended. Finally, it is important to note that running SecureDrop on a HTTPS URL would partly mitigate the leakage problem.

SD-01-011 IPTABLES configuration allows outbound traffic *(Medium)*

Tracked as issue #203 on GitHub.

The default *iptables* configuration deployed by the SecureDrop installation script does not restrict any outbound traffic. This allows an attacker who gains code execution privileges to connect to the open web and spawn connect-back shells.

Outbound traffic always needs to be monitored (as per OSSEC configuration). Even more importantly so, it has to be restricted for non-shell users like apache’s *nobody*. This is best achieved with the iptables *owner* module where each outgoing connection is tied

[6] Noreferrer http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html#link-type-noreferrer

to a specific user or otherwise dropped. A secure configuration is seen in the following example where only *james* user is allowed outgoing HTTP traffic:

```
iptables -A OUTPUT -o eth0 -p tcp --dport 80 -m owner --uid-owner james -m state
--state NEW,ESTABLISHED -j ACCEPT
iptables -A INPUT -i eth0 -p tcp --sport 80 -m state --state ESTABLISHED -j
ACCEPT
iptables -A OUTPUT -j DROP
```

SD-01-012 Flask cookies leak (server-side) session values *(Low)*

Tracked as issue #204 on GitHub.

The configured secret key which is required for using the Flask session feature is not used to encrypt the session values in the cookie but only to sign them. This means that the cookie can be decoded to show the values in plain text.

PoC:

```
>>> cookie_str =
"eJw9zMsOQ0AYQOFXaf61BVobSRcuGbVApC6Z2TSYMYohUYo23r3VRZcnOfneUPSUdZlgoL_
hkIMORF0qYpkVVYmSW2aTpZrsOtqTpDMvHCTTS8OxSCb6e9BKrsYSRHiGTYLiMZS3sW9Y9-
f8V6j5dVLhFInADlesxsfACU9-
jYQvkMBRrBHbUILUVbwIL57NFRKezztXthnnjIJeZu2DSdD2e97uX30cJrZ9AOFDPgE"
>>> zlib.decompress(base64.urlsafe_b64decode(cookie_str+b"="*(-
len(cookie_str) % 4)))
---
{"codename":
{"b":"Z2xhZCBhd2Z1bCBkaW50IG5vZWwgcGF0dHkgYmVudCBhd2FyZSAxOTYw"},
"csrf_token":{"
b":"NzQ5NjVhYWFmODQyY2U3OGQ4NjFmNmFmYTU5ZDA1OWI1MTYxMDg1ZQ=="},
"flagged":false,
"logged_in":true}
---
>>>
base64.b64decode('Z2xhZCBhd2Z1bCBkaW50IG5vZWwgcGF0dHkgYmVudCBhd2FyZSAxOTYw')
'glad awful dint noel patty bent aware 1960'
```

Two methods can be used to mitigate this leak. The first one is to implement one's own session interface that uses encryption for the (de)serialization, subsequently passing the resulting cookie to the app via the *app.session_interface* config. The second method is to exclusively store a session id in the cookie and create a server-side session store with files or a database.

Miscellaneous

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of those findings are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while the vulnerability is present, an exploit might not always be possible.

SD-01-003 Overly permissive Database privileges for "securedrop" user *(Low)*

Tracked as issue #193 on GitHub

The privileges given to the MySQL user accessing the SecureDrop database are overly permissive, therefore allowing an escalation of privileges to an attacker who has successfully performed an SQL injection attack. The source of this problem can be traced back to the installer files:

Example: https://github.com/freedomofpress/securedrop/search?q=GRANT+ALL+PRIVILEGES&ref=cmdform

Code:

```
echo "Setting up MySQL database..."
mysql -u root -p"$mysql_root" -e "create database securedrop; GRANT ALL
PRIVILEGES ON securedrop.* TO 'securedrop'@'localhost' IDENTIFIED BY
'$mysql_securedrop';"
```

Naturally, any given user should *only* be applied with those privileges that they might actually need. Generous dealings and granting of privileges can be destructive, as in the case of an attacker managing to spot an SQL injection vulnerability. As it stands now, the user with an unlimited set of permissions would be capable of compromising the machine via FILE and other SQL features (depending on the MySQL version)[7]. From what can be seen from the application's logic, the database user essentially needs a read-write access to one particular database rather than being awarded a "GRANT ALL PRIVILEGES" option.

Furthermore, a use of randomized or, at the very least, obfuscated table names and columns is advised in response to a possible discovery of an SQL injection vulnerability being discovered by a rogue party. This is a prevention mechanism that will raise the bar for the attacker, making an effort necessary to extract usable information considerably greater. Currently the setup picks the following database parameters:

Code:

```
DATABASE_NAME = 'securedrop'
DATABASE_USERNAME = 'securedrop'
DATABASE_PASSWORD = ''
```

[7] MySQL :: Privileges http://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html

Employing random database name, database user name and randomized elements in the table's column names and similar features whenever possible is a clear recommendation. While our tests did not identify SQL Injection vulnerabilities, a possibility that later versions become exploitable cannot be excluded, suggesting a strict least-privilege policy mandatory.

SD-01-004 Lax Permissions for *google-authenticator* Files *(Low)*

Tracked as issue #201 on GitHub

A small privilege misconfiguration problem was spotted on the App-Server, where the *admin1*-user on the given App-Server instance is able to modify his own emergency scratch codes. This should not be possible.

Example:

```
-rw-r--r-- 1 admin1 admin1 140 Dec 3 06:33 .google_authenticator
```

While this is an issue rated "low" in terms of severity, the best practice would definitely be to let the root user handle the emergency-login codes. Otherwise, a compromised SSH-account might be used to add an arbitrary amount of login codes without anyone noticing, thusly securing valid logins for future sessions without detection. Basically, it must be ensured that the google-authenticator file is root-owned and not writable for and by other users.

SD-01-005 Lax Permissions for Files inside the Webserver's DocRoot *(Low)*

The permissions for the files inside the document root folder ("*/var/www/<source/document>/securedrop*") are not changed after the default install, which signifies that they continue to be owned by the user that ran the install script. This should not be the case.

Once the SSH credentials of such user-owner get compromised, an attacker might be able to alter the source code of the SecureDrop installation. While changes to the web-server files will eventually get detected by the default OSSEC installation, a best practice recommendation is to guarantee that every file inside the document root folder of a web-server is writable to the root-user *only*.

SD-01-007 Considerations about TBB Configuration Settings *(Medium)*

Tracked as issue #196 on GitHub

In some scenarios, the connecting user might face the simple unavailability of the hidden service or similar networks woes that potentially eventuate in an unsolicited data leakage. The Gecko browser engine, foundation for the TBB and Tails' IceWeasel browser, applies automatic fixes to unresolvable URLs.

These "fixes" include prefixes and suffixes that are automatically added to URLs that do not seem to resolve properly. In consequence one might end up with, for instance,

having the URL *jgdevcded4x6dl37.onion* be "fixed" into either ***www****.jgdevcded4x6dl37.onion* or even ***www****.jgdevcded4x6dl37.onion.****com***. A highly skilled attacker might be able to misuse this feature to redirect TOR Hidden Services requests to actual WW-websites without the user knowing.

The "fix" applied by Gecko can be reverted by accessing the browser settings (*about:config*) and changing the following settings' values to empty strings:

- `browser.fixup.alternate.prefix`
- `browser.fixup.alternate.suffix`

A very tricky issue has been discovered in the Tails OS version 0.2.2 - the most recent version available during our tests. The distribution upgraded the default browser to IceWeasel 24, so that for the version 0.2.1, the 17.0.10 version was used. Contrary to the IceWeasel 17, the 24 version supports a feature called WebRTC. Said feature can be misused to unveil local IPs of the users without their consent and it was first demonstrated by the maintainer of the http://net.ipcalf.com/ website:

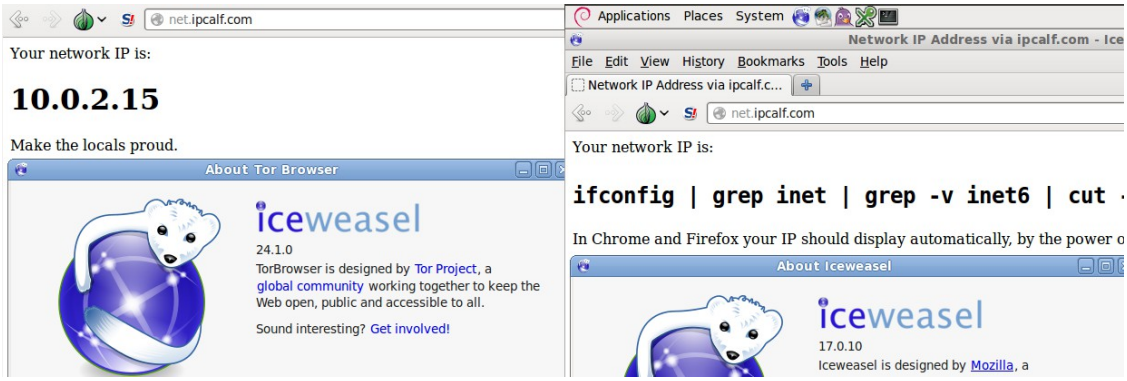

Fig.: Leakage of internal network IP in Tails 0.2.2 via WebRTC

To mitigate this issue, the WebRTC feature should be entirely disabled. In order to perform this, one has to access the IceWeasel's configuration editor (*about:config*) and implement the following setting:

Setting
`media.peerconnection.enabled: false`

Once this setting has been set to false, the WebRTC feature is completely disabled and local IP leakage cannot happen any longer. While this is not a critical data leakage, it might be of great interest to a strong adversary who seeks to collect important bits of information for further narrowing down of a target, eventually aiming for de-anonymization. The Tails developer has been informed about this issue, responded and will address the problem.

SD-01-009 Possible Attacks via unfiltered File-Names in ZIP-File Creation *(Low)*

Tracked as issue #197 on GitHub

SecureDrop puts uploaded files directly into a ZIP file which is then getting encrypted. A journalist then, upon reception, downloads the encrypted ZIP file to decrypt and extract it on the secure and air-gapped viewing machine. The filename of what is extracted stems from what was being used in the upload request for the ZIP file. This can be controlled by an attacker.

Affected Code:

```
fh = request.files['fh']
[...]
zip_file.writestr(fh.filename, fh.read())
```

Filenames that include "/" (eg. "*folder/name.docx*") will create folders inside the archive because they are interpreted as paths. An attacker could also use "../" and absolute paths to traverse through the file-system, even though most unpackers will warn, prohibit or just strip away path traversal. Regrettably, the latter is not guaranteed and so is the behavior of other special characters. What is to be kept in mind is that filenames that are too long may also crash the unpacker.

It is suggested to pass the filename through a filter first, assuring that only ASCII characters are allowed and replacing any non-ASCII and special character sequences (such as "/") with safe characters like "_".

SD-01-010 Denial-Of-Service for Source via UTF-8 in Journalist-Message *(Medium)*

Once a journalist starts communicating with a source, message objects are being created and secured by a strong encryption. The journalist can nevertheless disable the interface for the source by adding specially formed Unicode characters to the message body. The application will crash upon parsing those and destroy the source's ability to log in with the formerly chosen pass-phrase. An example message was posted and can be tested with a use of the following pass-phrase:

"baird wrath bohr vivid malady flam chris pogo"

The following stack-trace is being generated upon message parsing:

```
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1] mod_wsgi (pid=3034): Exception
occurred processing WSGI script '/var/www/securedrop/source.wsgi'.
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1] Traceback (most recent call last):
...
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1]       {% extends "base.html" %}
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1]   File
"/var/www/securedrop/source_templates/base.html", line 26, in top-level template code
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1]       {% block body %}{% endblock %}
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1]   File
"/var/www/securedrop/source_templates/lookup.html", line 19, in block "body"
```

```
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1]       <blockquote class="message">{{
msg.msg }}</blockquote>
[Wed Dec 04 14:09:36 2013] [error] [client 127.0.0.1] UnicodeDecodeError: 'ascii' codec
can't decode byte 0xc2 in position 2582: ordinal not in range(128)
```

To prevent this from happening, it must be assumed that all messages exchanged between journalists and sources can in fact contain valid and invalid UTF-8 characters. It is recommended to decode the text to avoid application crashing.

Fix Recommendation:

```
crypto_util.decrypt(
        g.sid, g.codename, file(store.path(g.sid, fn)).read()
).decode('utf-8')
```

Conclusion

First and foremost conclusion is that this penetration test against SecureDrop did not yield any critical vulnerabilities. The majority of the findings was based on small mistakes in the code, forgotten security measurements against browser-specific attacks, lack of certain HTTP headers and several “medium-severity” weaknesses that provided a first step for an attack, yet left no room for the second and detrimental step. SecureDrop presented itself as a very well-hardened application with a limited attack-surface and a small code-base.

Several attack scenarios were being discussed with the SecureDrop team during the test and did not end up in separate tickets, particularly when the attack was rather theoretical, hard to prove as fully functional or requiring a complex set-up, social engineering or human failure. A selection of these is posted below.

- Traffic analysis / confirmation attacks partly de-anonymizing the source based on the fixed size of the HTTP response body for the SecureDrop static pages. Possibility to add random padding to the website’s HTML. The application might consider instrumenting randomized data inside HTML comments or data URIs to be able to obfuscate the response body on the wire.
- Traffic analysis / confirmation attacks partly de-anonymizing the source based on a very unique and possibly known file-size of the leaked document. Possibility to add random padding to the website’s POST request for uploads via *<textarea>* / JavaScript. The application might consider instrumenting CSP headers and JavaScript in order to scramble form’s request body and add additional noise to obfuscate the upload on the wire.
- Local compromise of the source by a rogue journalist providing a malicious PGP key for download, for instance an OpenPGP+PDF chameleon containing both the key and the active PDF data compromising Adobe reader or Evince. The application should sanitize or validate PGP keys and scrub any of the possibly hidden data.

- Social engineering performed by a potentially rogue or bribed journalist tricking the victim into unveiling their identity via the messaging system. The application must clearly state that the journalist might not be trustworthy enough for the source to blindly follow instructions sent via the messaging system.

In conclusion, we believe that SecureDrop's greatest challenge lies not in creating a technically secure application, communication channels and server architecture but rather in getting technically less proficient users and whistle-blowers to benefit from the system without risking to leak their identity. While this might have seemed hardly possible in the application's original state, it has already demonstrated significant progress and improvement when compared to what was described by Czeskis et al.. SecureDrop is on its way to reaching a primary goal of providing an exceptionally strong system, focused particularly on security and anonymity aspects. It is now reaching a moment when developing ways to work on accessibility and installation ease are vividly important. Discovering the best ways towards educating users to securely and safely deal with the documents they intend to submit or receive should be framed as a main concern.

Cure53 would like to thank Trevor Timm, James Dolan, Garrett Robinson and the SecureDrop Team for their support and assistance during this assignment.