DeadDrop/StrongBox Security Assessment

Prepared On:

August 11, 2013

Performed By

Alexei Czeskis <aczeskis@cs.washington.edu>
David Mah <mahh@cs.washington.edu>
Omar Sandoval <osandov@cs.washington.edu>
Ian Smith <imsmith@cs.washington.edu>
Karl Koscher <supersat@cs.washington.edu>
Jacob Appelbaum <jacob@appelbaum.net>
Tadayoshi Kohno <yoshi@cs.washington.edu>
Bruce Schneier <schneier@schneier.com>

University of Washington
Department of Computer Science and Engineering
Technical Report: UW-CSE-13-08-02

Table of Contents

EXECUTIVE SUMMARY **3**

1.) ASSESSMENT SCOPE AND GOALS **5**

2.) BACKGROUND: DEADDROP THREAT MODEL AND GOALS **5**
- 2.1) ASSETS 6
- 2.2) ATTACKERS 6

3.) DEADDROP DESIGN ASSESSMENT **7**
- 3.1) USAGE OVERVIEW 7
- 3.2) INFRASTRUCTURE OVERVIEW 7
- 3.3) DESIGN ANALYSIS 9

4.) OBTAINING DEADDROP CODE/DOCUMENTATION **11**
- 4.1) ISSUES WITH SOURCE CODE ACCESS CONTROL 11
- 4.2) ISSUES WITH BRANCHING AND VERSIONING 12
- 4.3) TYPOSQUATTING 12

5.) DOCUMENTATION ASSESSMENT **12**

6.) IMPLEMENTATION ANALYSIS **14**
- 6.1) USABILITY ISSUES 14
- 6.2) ATTACKS 15
  - *Leaking the Source's Identity* *15*
  - *Denial of Service Against Journalist* *17*
  - *Codeword collision* *17*

7.) DEPLOYMENT AT *THE NEW YORKER* **18**
- 7.2) TEST SUBMISSIONS 19

8.) USAGE ISSUES **19**
- 8.1) ATTESTATION OF SERVER-SIDE CODE 19
- 8.2) ANONYMITY BEYOND DEADDROP 20

9.) CONCLUSION AND RECOMMENDATIONS **20**

A.) APPENDIX: DOCUMENTATION ANALYSIS **22**
- A.1) THREAT_MODEL.MD 22
- A.2) README.MD (INSTALLATION INTRUCTIONS) 24

About Us

We are a group of independent academic and industry security researchers who have no stake in the outcome of this assessment. We performed this security assessment free-of-charge and independently of the developers of this software or of any party using this system. This audit was done purely as a public service. This is a technical report; while we believe that the technical results are correct, the surrounding discussion may not reflect all of the authors' personal opinions.

Executive Summary

Several media articles[1] have recently reported on a newly developed system called *DeadDrop*, which is specifically designed to provide anonymous communication between individuals (sources) and journalists in the face of very powerful adversaries (*e.g.,* oppressive regimes)[2]. Initially designed and developed by the late Aaron Swartz, DeadDrop comes at an interesting time when leaked documents are being discussed in the media[3]. If DeadDrop takes off, it could significantly change the way that journalists and individuals communicate anonymously across the world, which could have a significant impact on a wide variety of reporting.

DeadDrop has the goal of being secure against adversaries that can (and there is precedent for this) monitor communications of the press and pursue (and punish) whistleblowers. Given that DeadDrop has already been deployed by at least one prolific media organization, *The New Yorker*[4], we wanted to understand whether DeadDrop actually provides the security properties necessary for anonymous communication in the face of such strong adversaries.

We acknowledge that DeadDrop can be used for many purposes, ranging from those that some might argue as ethical and legal to those that some might argue as unethical or illegal. We observe that similar statements have been made about other types of security technologies in the past. For example, in the past people have argued that making email encryption broadly available was unethical -- despite its clear benefits in many legitimate scenarios -- because email encryption could be used for illegal purposes. In this document, we take no stance on the ways in which DeadDrop *should* be used, but rather focus on the system design, implementation, and usability in the ways that the system does or does not support anonymous communication.

Generally speaking, DeadDrop is designed to work as follows: a media organization deploys DeadDrop on its servers. Individuals (*sources*) who want to anonymously communicate with journalists visit the organization's DeadDrop deployment page and are shown four random *codewords*, which the source is supposed to remember. The source is then shown a page that allows him or her to submit messages and documents. The DeadDrop system is designed to encrypt all messages and documents in such a way that only the journalists are able to decrypt them. The journalists can communicate back to the source by leaving messages in the DeadDrop application, which the source can view by visiting the DeadDrop page as before and entering his/her codewords. Messages for the source are encrypted and are designed to only be decryptable using the codewords. Conceptually, DeadDrop creates anonymous mailboxes that the source and journalist can use to communicate with each other. DeadDrop uses several techniques designed to prevent the journalist from learning the source's identity (including the source's IP address or location). Similarly, DeadDrop is designed to safeguard the privacy and confidentiality of the source's submitted messages and files in case of global internet monitoring and even physical removal of the DeadDrop servers.

---

[1] http://www.guardian.co.uk/world/2013/may/17/new-yorker-strongbox-aaron-swartz-data-privacy
[2] http://deaddrop.github.io/
[3] http://www.washingtonpost.com/blogs/wonkblog/wp/2013/06/12/heres-everything-we-know-about-prism-to-date/
[4] www.newyorker.com/strongbox

We performed a rigorous technical assessment of the DeadDrop documentation, design, and implementation, analyzing the system for resilience against a variety of possible attacks. We also deployed an instance of DeadDrop[5] and analyzed the usability of the system both from the source's and the journalist's point of view.

The conclusion of our analysis is that many of the technical properties of DeadDrop are decent; however, we do not believe that DeadDrop is yet ready for deployment in an ecosystem with nation-state capable adversaries and non-expert users. The lack of software versioning, reliance on VPN, the errors in the installation and deployment documentation, leaking of document metadata, and lack of anonymity best practices all contribute to our reluctance for suggesting that DeadDrop is ready for mass deployment.

Additionally, the usability of the system is sometimes lacking, potentially leading to insecure use. For example, DeadDrop requires a fair amount of technical sophistication on behalf of journalists (such as being able to use the GPG encryption software)[6] and sources (such as being able to sanitize the metadata in the submitted documents). We believe that this lack of usability may lead to failures in anonymization. We enumerate the usability pitfalls we found, as well as suggested remediation approaches in our report.

We also briefly analyzed *The New Yorker*'s deployment of DeadDrop, called "StrongBox", and provide feedback on the deployment. Though we were only able to perform a blackbox analysis, we identified a number of issues. For example, *The New Yorker* does not use HTTPS for the page advertising StrongBox. Additionally, we believe that StrongBox was not installed using the best practices given in the DeadDrop installation document. For example, the StrongBox deployment leaks HTTP server version – something that is specifically forbidden in the installation documents. We believe these issues should be fixed immediately.

As part of this assessment of Strongbox, we submitted specially crafted documents in order to evaluate whether *The New Yorker* was indeed checking submissions correctly (on an air-gapped system) if at all. The documents contained requests for The New Yorker staff to reply to our documents, and instructions on how to do so. Unfortunately, after more than 9 weeks, we received no reply from *The New Yorker* and are unable to speculate on whether they received the documents or not. To our understanding, StrongBox staff was repeatedly informed about our submissions via a media contact, and were explicitly asked – by that media contact – to look at the submissions and reply.

We hope that this analysis will prove useful to users of DeadDrop (both anonymous sources and journalists) and to the DeadDrop developers. Some of the issues we identified are already being addressed. We invite subsequent, even deeper analyses into the technical properties and usability properties of the system.

---

[5] DeadDrop does not have version numbers (see Section 4.2). However, the corresponding hash of the git commit we deployed was `a5727a9afb35718ddfb7df0f37521795ccf973c9`. The hash of the commit for the DeadDropDocs repository that we used was `02b4728e1577249f5c0b957bb44169fc60b0df2c`.

[6] http://arstechnica.com/security/2013/06/guardian-reporter-delayed-e-mailing-nsa-source-because-crypto-is-a-pain/

1.) Assessment Scope and Goals

This assessment has several interrelated stages. These stages address interrelated properties that must all be met in order for the resulting DeadDrop deployment to provide sufficient security properties for uses.

- **Design Analysis:** First, we consider the design of the DeadDrop system, as reflected in the design documents available on GitHub on June 4th, 2013. During the assessment of the design, we examine the threat model -- including the attackers, threats, and risks to the DeadDrop system.
- **Implementation Analysis:** Second, we analyze the DeadDrop implementation, checked out from GitHub on June 4th, 2013. As part of the implementation audit, we examine the DeadDrop code base in search of commonly found vulnerabilities, misconfigurations, incorrect use of cryptographic primitives, poor sources of entropy and other errors that might lead to security risks.
- **Documentation and Deployment Analysis:** Third, we consider how DeadDrop is being developed, maintained, documented, and deployed. Specifically, we analyze whether the current practices encourage or discourage secure use and deployment by potential DeadDrop clients. We consider the deployment of DeadDrop by *The New Yorker*, which is branded with the name *StrongBox*.
- **Usage Issues:** Fourth, we examine the broader security and privacy issues surrounding the usage of the DeadDrop system. We consider the role that DeadDrop plays in the broader context of secure and anonymous communication between sources and journalists and whether sources can actually expect anonymity even by using DeadDrop.

At each step we will provide remediation advice and guidance when possible.

Much of DeadDrop's security is built on top of other secure services: *e.g.*, the Tor anonymity service and the GPG encrypted and authenticated email system. We will not be reviewing those systems, and will be assuming their security in our analysis of DeadDrop.

2.) Background: DeadDrop Threat Model and Goals

According to the documentation available on June 4th, 2013, the main goal of DeadDrop is to provide an environment for individuals (*sources*) to anonymously communicate with the media (*journalists*). The name *DeadDrop* refers to the practice long used in a variety of communities as a method of communication and physical item transfer that does not require two entities to ever physically meet (thus protecting each individual's identity). For example, two individuals might agree that one person will place a secret message in an inconspicuous book in a library on a particular day. The other person would then retrieve the message from the book at a later time. DeadDrop was designed to mirror this communication concept in the digital space. The foremost goal of DeadDrop is to provide anonymous sources and journalists a secure, anonymous, and private platform of communication. We review the system assets and attackers below:

2.1) Assets

The main assets of the system are:

- Identity of the individuals (sources) submitting data to journalists. Anonymity of the source should be preserved.
- Integrity and confidentiality of the communication between sources and journalists. The data communicated between source and journalist should be secure.
- Availability and usability of the system. Attackers should not be able to easily make the system be unavailable (*e.g.*, by making the system go “down”) or unusable (*e.g.,* by making the system be too slow).

In addition to the above primary assets, we also consider a variety of secondary assets such as plausible deniability for sources (*i.e.,* that they are not the source of a leaked document).

2.2) Attackers

The system attempts to be resilient against a wide range of attackers:

- Network attackers. Such attackers who are able to monitor, record, and modify network traffic. Examples of such attackers might be Internet Service Providers, individuals with wiretap powers, or just individuals sniffing WiFi in cafes.
- Individuals with Physical Access to DeadDrop Servers. Some of the employees working for the media organization hosting the DeadDrop service may be less trustworthy than others. Some of these employees may try to gain access to the DeadDrop documents. Additionally, either through theft, subpoenas, or accidental disposal of the computers, the DeadDrop machines (as well as cryptographic keys and passwords) might get into the hands of adversaries.

In addition to the above primary attackers, in our analysis we also consider attackers that might try to submit malicious documents to DeadDrop and attackers that attempt to compromise the DeadDrop source code.

The system does not aim to provide any security if the source's computer is compromised. Neither does the system attempt to protect against attackers that might modify the code upon which DeadDrop depends -- such as Tor.

3.) DeadDrop Design Assessment

The DeadDrop documentation (on June 4th, 2013) contains a description of the overall system design. We summarize the design briefly before commenting on the security properties.

3.1) Usage Overview

To submit data, the source performs the following steps:

1. The source visits a media organization's website and finds the anonymous submission page. For example: http://www.newyorker.com/strongbox/. This link instructs the source to download the Tor browser bundle[7] and gives the source a .onion address[8] to visit.
2. The source downloads the Tor browser bundle, if he/she doesn't already have it installed, and visits the provided .onion address. The source is then shown four randomly generated words that he/she is supposed to memorize[9]. These are the codewords (or password) that the source will use in order to maintain a single (though anonymous) identity on the website.
3. The source is then presented an interface whereby he/she can send messages and files to the journalist.
4. The source can at any time return to the .onion site, submit the codewords he/she received earlier and be able to check for replies from the journalist or submit more documents and messages.

To retrieve documents, the journalist performs the following steps:

1. The journalist accesses an internal website which alerts him/her to the existence of new submitted data. The data is encrypted using the journalist's GPG key.
2. The journalist downloads this encrypted submitted data onto a flash drive and transfers the flash drive to the secure viewing station (see below), inserts the flash drive with encrypted data and the flash drive with his/her GPG key, decrypts the data, analyzes it, and destroys the unencrypted data.
3. The journalist returns to his/her desktop, accesses the internal website and sends a reply to the source. This reply will be placed in the source's mailbox (see step 4 above).

3.2) Infrastructure Overview

The media organization sets up an infrastructure consisting of the following parts:

- The *source interface* -- a computer running the interface that the source will use to "create an account", send messages and files to the journalist, and receive replies from the journalist. More technically, this is a Python script running a web server that is listening as a Tor hidden service. The computer running the source interface should be

[7] https://www.torproject.org/projects/torbrowser.html.en

[8] An address that must be resolved through the Tor network. For more, see the Tor documentation at https://torproject.org

[9] The source can write the codewords down, but then give up some plausible deniability if those words are physically discovered in the source's possession by the adversary.

physically located in the news organization and be behind the corporate firewall. The Tor hidden service should be allowed to punch through the firewall.

- The *journalist interface* -- a computer running the interface that the journalist will use to receive files and messages from the source and reply to them. More technically, this is a Python script running a web server. The computer running the journalist interface should be on a different machine than the one running the source interface and should also be behind the corporate firewall. The source interface and journalist interface communicate with each other over sshfs.
- The *secure viewing station (SVS)* -- a machine that the journalist will use to decrypt files from the source. This machine should not be connected to the network, has no hard drive, and is booted off of a LiveCD.
- Additional infrastructure: there is much additional infrastructure not mentioned above, but is listed as being required or recommended (the documentation appears unclear) in the documentation. For example, the media organization is instructed to set up a monitoring server and a VPN.

Figure 1 summarizes the DeadDrop data flow.

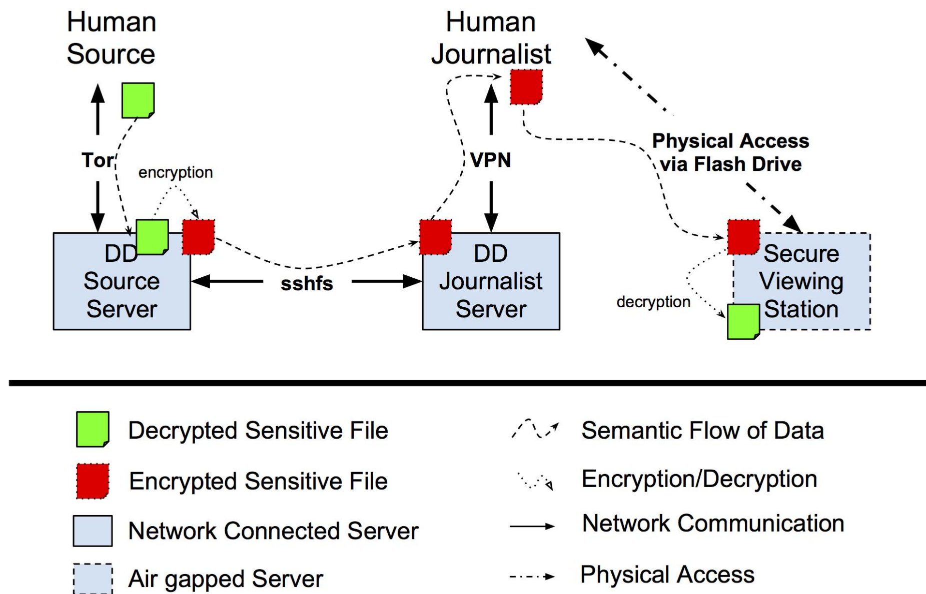

Figure 1: Simplified Summary of the DeadDrop data flow. The letters *DD* correspond to the DeadDrop software.

3.3) Design Analysis

For the most part, we found the DeadDrop design to be reasonable in achieving its stated goals. However, we also found a number of potential concerns with the DeadDrop design and list them below:

- It is possible for sources to submit data over a non-anonymous relay by using services like tor2web[10] or onion.to[11]. For example, we have successfully tested communicating with The New Yorker's DeadDrop deployment through: http://tnysbtbxsf356hiy.onion.to/ and https://tnysbtbxsf356hiy.tor2web.org/ without using the Tor browser. Both of these methods compromise anonymity to a network attacker because the DNS requests are sent in the clear. Either service might also keep logs, and in the case of onion.to, all HTTP data is unencrypted, allowing for the connection to be sniffed or man-in-the-middled. DeadDrop should take measures to prevent access not over Tor (see recommendation section). As users may not always understand the protections offered to them by Tor and might try to use these non-Tor services, we believe it is the duty of DeadDrop to help users avoid these type of mistakes.

- The system takes no effort to scrub documents of obvious identifying information and may leak sources' (or author's) identities through file metadata. For example. Consider a scenario where Alice is uploading a document she obtained from Bob, and the document metadata suggests that Bob is the author and also includes additional metadata. If Bob gave that version of the document to Alice and Charlie, then the existence of that data -- combined with information from Bob -- could reveal Alice as a potential source of the data. We provide a concrete experimental analysis in the Implementation Section and provide remediation advice in the Recommendation section.

- The codewords that sources are supposed to memorize are not easily memorizable. For example, in actual experimentation, we encountered the following words: *hoorahs weltering emulous televising* and *baguios dipnoans sadhu stink.* Additionally, the included word list contains some words with characters which may be hard to type on a standard American keyboard (e.g., *torchiËres*). There are 219 words with a non-ASCII character in them, meaning that there is a 0.7% chance of getting a non-ASCII word. Combined, it is likely that the source will have to write down the codewords. This, however, means that the if the source loses plausible deniability as he/she can potentially be found by the adversary in the possession of the codewords. As of this writing, a ticket has been opened to fix this issue, but no action has been taken.

- There are approximately 129,000 words in the dictionary, which is equivalent to 17 bits of entropy. Since the code phrase consists of 4 codewords, the total code phrase consists of 68 bits of entropy. This is far too little to protect against strong attackers. For example, if the attacker is able to obtain access to the DeadDrop hard drive, then the attacker will be able to gain access to the mailboxes. This will allow attackers to read the messages from the journalists to the sources (but not the submitted documents).

[10] http://tor2web.org/
[11] http://www.onion.to/

Attackers can buy off-the-shelf machines (such as https://products.butterflylabs.com/50-gh-s-bitcoin-miner.html) that will brute-force the entire codeword space in 10 weeks for $2.5 million. We suggest the use of alternate hashing techniques in the recommendation section to further protect against such offline attacks. As of this writing, a ticket has been opened to use a slower hashing algorithm, but no action has been taken.

- The source must be careful to not contact the DeadDrop deployment site from a machine that might record keystrokes or take screenshots. This means that the source must somehow remove documents to be leaked from their original environment. This might be tough in some environments. The DeadDrop interface should give some guidance on this.

- The Secure Viewing Station (abbreviated as SVS in the documentation) is booted off of a LiveCD and not connected to the network. Unfortunately this makes it difficult for the SVS to be up-to-date and fully patched. It is conceivable that attackers might submit malicious files that exploit unpatched vulnerabilities in the SVS. For example, the attacks might delete the secure keystore USB drive (preventing further interaction with DeadDrop), exfiltrate data through the external hard drive, or attack the DeadDrop hidden service through the external hard drive.

- The design of the SVS as an air-gapped machine might cause journalists to bypass the SVS entirely and just decrypt sensitive files on their desktops – potentially without any kind of secure delete afterwards. This means that if the journalist's hard drives are ever compromised by the attacker (physically or remotely), the sensitive files submitted by the source might be at risk of access, deletion, or modification. Furthermore, if the submitted documents are deliberately malicious (as might be done by attackers), these documents might compromise the journalist's machine. An alternative might be to recommend that journalists view sensitive files by booting their desktop into the Tails anonymity VM[12]. More effort should be put into making the SVS more usable for journalists while also being secure.

- VPN connections expose the network (and physical) locations of the two end-points. If the DeadDrop servers for a media organization are hosted off-site, then by using a VPN to connect to the DeadDrop servers, journalists would expose the location of those servers -- making it easier for the attackers to physically compromise the machines. Specifically, if DeadDrop is operating in a country where the government may liberally seize computer equipment, then the use of a VPN will put that equipment at risk. Therefore, journalists should connect to the journalist interface over a Tor hidden service rather than a VPN. This isn't an effort to provide security through obscurity, but rather add an extra layer of defense by making it harder to figure out which equipment to seize and monitor. We'd like to note, however, that using a Tor hidden service will not fully mask the location of the DeadDrop servers, as the source interface will stand out on the

[12] https://tails.boum.org

network as a Tor Hidden Service if it is idle. It will also likely stand out if it is heavily loaded for submission (*e.g.,* if a source or attacker uploaded a DVD). The journalist hidden service will stand out when the journalist is communicating with it. However, these types of sophisticated network flow analyses are more difficult than attacking the VPN directly.

- The systems is vulnerable to a denial of service attack whereby adversaries may "leak" untrue data in mass quantity. This type of attack would overwhelm journalists as they would be continuously checking the DeadDrop submission. Recall that checking each submission requires much manual effort such as putting data on USB disks, physically taking the disks to the secure viewing station, and decrypting files. The attack would then likely prevent journalists from being able to legitimately use the system.

- The system does not provide a method for journalists to be notified when a submission occurs. This means that journalists would either have to continuously check the journalist interface for new submissions or face the chance that they might miss an important yet timely submission. This detracts from usability and decreases the chance that journalists will use this service. Similar concerns exist for the source (no notification when receiving a reply from the journalist). It is incredibly important for sources to receive timely notifications as the source puts himself/herself at increased risk every time they visit the DeadDrop server, though we acknowledge that there may be no way of doing this securely.

- Encryption of the source's data is done on the server. A more secure approach would be to do data encryption on the client using the journalist's public key. This would help to protect against attackers that might compromise the DeadDrop server. Perhaps such an encryption client could be shipped with the Tails anonymization VM or as a plugin in the Tor Browser Bundle. Similarly, the journalist keys could also be shipped in the Tails VM so that sources don't need to authenticate the journalists' keys.

4.) Obtaining DeadDrop Code/Documentation

The code and documentation are hosted on *GitHub* at the URL: https://github.com/deaddrop/. GitHub forces that the fetching of data is done through TLS, which *usually* assures confidentiality, integrity, and authenticity of the data. Because DeadDrop is such a security critical service, it's imperative that media organization take extra care to check the authenticity of the code (*e.g,* to protect against man-in-the-middle attackers).

4.1) Issues with Source Code Access Control

The last publically known breach of GitHub (that we could find) occurred on March 4th, 2012: https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation. Since DeadDrop "joined" the GitHub prior to the breach (on Nov 11, 2011), DeadDrop should take care to make sure that contributing members have rotated their SSH keys following the breach and that no

code modification occurred during the breach. Only authorized individuals should have GitHub credentials, and those credentials should be kept securely.

4.2) Issues with Branching and Versioning

It appears that the current structure of the code and documentation on GitHub is not organized into any versions. Specifically, there is no mention of *stable* or *experimental/beta* versions. This makes keeping up with critical changes difficult. This organizational structure has at least several downsides. First, clients of DeadDrop might miss a critical update (mistaking it for a typo fix -- as many updates are just that). Second, DeadDrop clients might download and install code that is not thoroughly tested -- introducing potentially critical bugs into a sensitive environment. Finally (this is likely an artifact of the code organization), there are no cryptographic hashes of the code -- making it difficult for outside parties to vouch for the security of a particular version (we note that GitHub does provide hashes that correspond to particular commits, but we believe that additional hashes should be provided over specific versions of the code).

4.3) Typosquatting

Developers for DeadDrop should be careful to monitor GitHub and other source code repositories for appearance of evil clone projects that might confuse potential DeadDrop users. That is projects with names such "daeddrop" and "deadrop" should be examined to make sure they don't imitate DeadDrop in scope. Similar care should be taken with projects bearing a name resembling StrongBox.

5.) Documentation Assessment

The documentation for DeadDrop design and installation is located on GitHub at the following URL: https://github.com/deaddrop/DeadDropDocs. We found much of the documentation to have errors with respect to the commands that needed to be typed and inconsistencies (instructing us to install a package from the network while previously stating that a machine was supposed to be off-line). It's not clear who will be consuming the documentation, but the sheer length (436 lines) of the README document might seem intimidating to some system administrators (who would presumably be the ones setting up the system). Additionally, the design and deployment diagrams seem overbearing and confusing -- giving (in our opinion) too much technical detail. We provide a copy of the deployment diagram in Figure 2 and a copy of the design diagram in Figure 3. Generally, we suggest making several levels of documentation -- a general system overview and an install guide. We went through all steps in the installation procedure detailed in the README document and found many errors. Please see Appendix A. for a detailed assessment of the documentation.

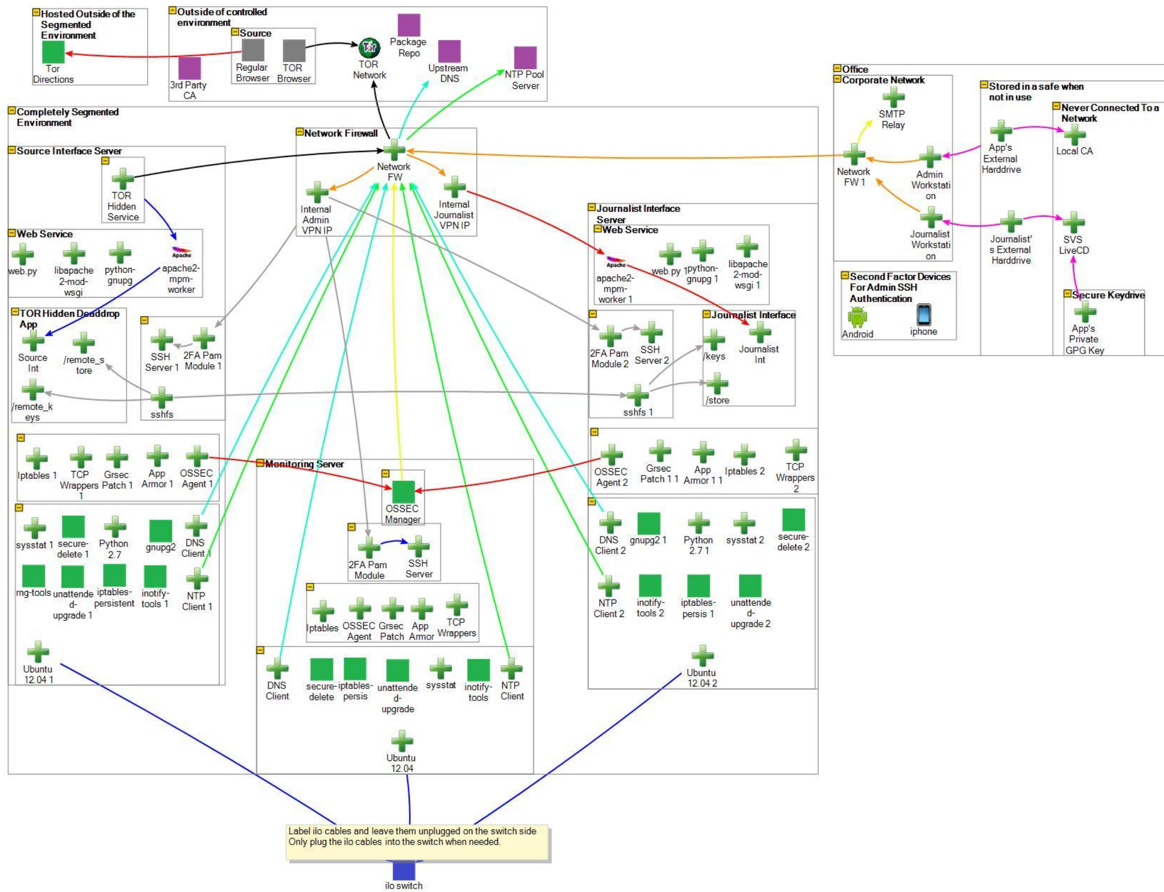

Figure 2. Deployment diagram provided by the DeadDrop documentation.

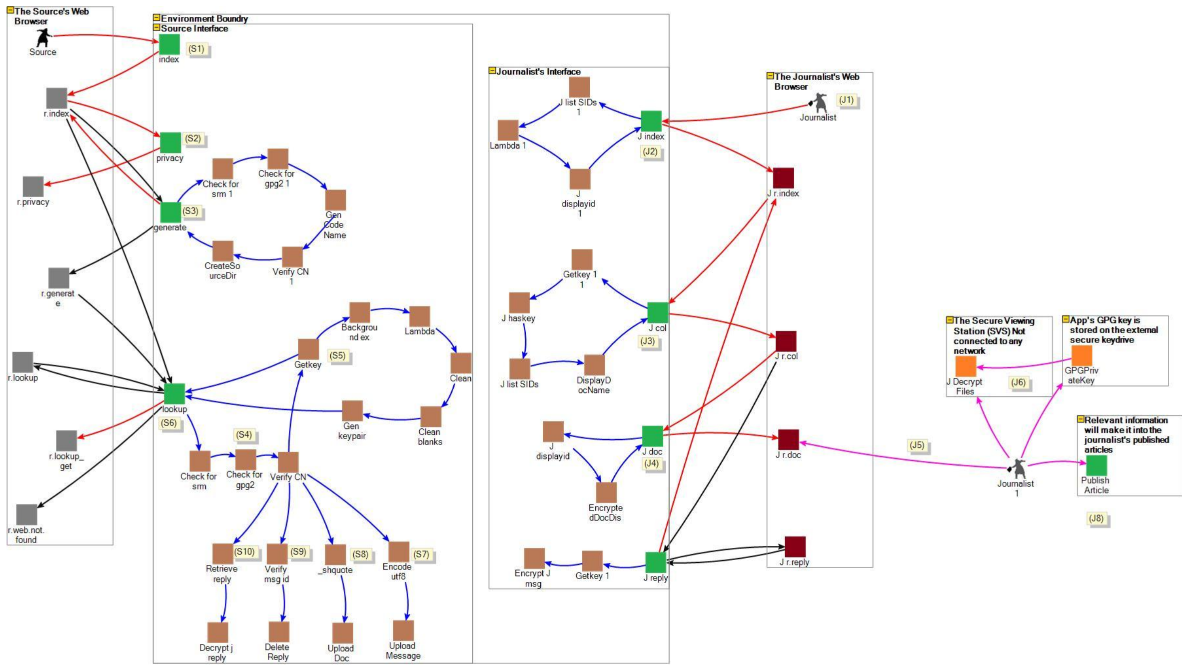

Figure 3. Design diagram provided by the DeadDrop documentation.

6.) Implementation Analysis

We attempted to deploy a version of DeadDrop on our internal network by closely following the provided instructions. After approximately 30 person hours, we concluded that the combination of incorrect directions, errors in provided scripts, and complicated design make it unreasonable for regular system administrators to set up DeadDrop correctly. Though we were able to reverse engineer and fix many bugs, we were unable to make the provided puppet setup scripts work. In the end, we deployed a functional system, but it lacked some of the auxiliary components such as the VPN, monitoring server, and some hardening modifications that were supposed to be imposed by puppet. However, we believe that this setup is consistent with what one might expect if a systems administrator were to try to configure DeadDrop for an organization (see *The New Yorker* Deployment section below). We further believe that that this setup does not change the evaluation of the system given the expected threats we evaluate. The additional steps given in the install steps help to protect DeadDrop in case a flaw is found in the underlying operating system, web browser, or any other supporting infrastructure. We do not study such attacks.

6.1) Usability Issues

We identified a number of usability issues in attempting to use our DeadDrop deployment. We enumerate key observations below:

As the Journalist:

- Journalists have to use command line gpg instructions to decode files on the Secure Viewing Station. Following such instructions can be tasking; we wrote our own scripts to handle the gpg decryption process. Some individuals might have difficulty actually using gpg from the command line. If journalists are not able to easily decrypt and view documents, it is likely that important documents might be uploaded but not accessed in a timely manner.

- Once a message is decrypted by gpg, a data file is created with no extension and no obvious indication on the file type. We had to use the `file` linux utility to discover the file type. Journalists should not be required to perform such manual command-line operations as this decreases the chances that the journalists would be willing to use the system and review files and timely manner. We later discovered that the file name and extension were coded into the gpg file (viewable by using the “-v” flag), but we did not find this anywhere in the DeadDrop documentation.

- There is no way for journalists to delete document submissions via the journalist interface. After several submissions, we found the interface to be somewhat cluttered -- making it difficult for us to keep track of particular submissions. Furthermore we missed several important submissions amid the clutter of spam submission. We expect that journalists might also miss important submission or experience similar frustration and be reluctant to check the journalist interface (or use the system at all).

As the Source:

- The codewords displayed to us were obscure (e.g, *baguios dipnoans sadhu stink*). We had to write down the codewords since we could not remember them (even among several people). For real sources, this may mean the lack of plausible deniability if the codewords are ever discovered by the adversary.

- Since there is no reliable indication of a successful file upload, we had no idea whether or not our files had been uploaded to DeadDrop successfully. As a result, we accidentally terminated large file uploads early by closing the browser, thinking that the upload had finished. This inability to determine whether a file has been successfully uploaded is a concern for several reasons. First, a source might similarly terminate the uploading of a document early -- thereby preventing the journalists from accessing the full contents even though the source might have gone to great lengths (and potentially endangered him or herself) to obtain the documents. Second, the source may attempt to upload the same document multiple times, thereby creating additional work for journalists -- who, if given too much work, may be less inclined to use the system.

6.2) Attacks

We examined the DeadDrop application source code for a variety of standard attacks. Our analyses did not uncover conventional vulnerabilities, such as user input sanitization, injection attacks, cross-site scripting, cross-site request forgery, buffer overflows, incorrect use of cryptographic primitives (bad cyphers, bad randomness sources, short key lengths), and several other attacks.

Nevertheless, the following attacks did succeed:

Leaking the Source's Identity

First, we experimented with leaking data to our DeadDrop deployment. We are not aware of the ways that actual leaked documents are submitted, but we assume that this way of leaking data is at least plausible. For all cases we tried to leak an anti-government manifesto: the Declaration of Independence of the United States of America (DoI). Here is a summary of the leaked data:

- Message 1 (Photo): A picture (JPG) of the DoI being displayed on a screen. The picture was taken with a smartphone running Andriod 4.2.1 with GPS location enabled (the default on many smartphones). We leaked the picture without any modification.

- Message 2 (PDF of Photo): We created a PDF document containing the image above by exporting the image from message 1 to PDF via the OS X Preview application. We then leaked this PDF.

- Message 3 (Word document of Photo): We created a Microsoft Word document using Microsoft Word for Mac 2011 (version 14.3.5) by pasting the image from message 1 into a new Word document. We leaked the Word document.

- Message 4 (PDF of Webpage): We used the “export as pdf” feature of the Google Chrome browser (Version 28.0.1500.44 beta for Mac) to save the DoI as a PDF. We leaked this PDF.

- Message 5 (Screenshot of Webpage): We used the screenshot feature of Windows XP with SP3 to take a screenshot of the DoI being shown in a browser. We saved the screenshot in a MSPaint (version 5.1, build 2600) BMP (as instructed by most of the on-line how-tos). We leaked this BMP.

We summarize the data we were able to recover about the source (as a journalist using the journalist interface) in the following table:

Message #	*Author’s Name*	*Author’s OS*	*GPS*	*Phone Type*	*Date/ Time*
1 (Photo)	no	no	**yes**	**yes**	**yes**
2 (Photo in PDF)	no	**yes**	no	no	**yes**
3 (Photo in Doc)	**yes**	**yes**	**yes**	**yes**	**yes**
4 (PDF of Webpage)	**yes**	**yes**	no	N/A	**yes**
5 (Screenshot)	no	**yes**	no	N/A	no

As the table above shows, we were able to extract at least some privacy leaking metadata from each file -- with the Microsoft Word Document being the worst offender. Without knowing exactly how sources will leak documents, it’s hard to tell what type of metadata might leak about their identities. However, in order to proactively protect the source’s identity, the DeadDrop application should incorporate measures to strip file metadata as it is submitted from as many file types as possible -- we envision this being simple for many file formats.

Metadata might also leak information about the history of a document. That is, the metadata might record that a document was created by Alice, modified by Bob, and sent to Carol. If Carol leaks the document Bob might be contacted by the journalist or by adversaries and pressured to confess that he sent it to Carol. Therefore metadata could pose a risk to the source even if it does not contain the source’s name.

Denial of Service Against Journalist

We investigated several usability attacks, which adversaries might launch in order to disrupt and discourage journalists from using DeadDrop. These attacks could be used, for example, by organizations that fear that their data might be submitted to DeadDrop.

Continuous File Submission

We wrote a simple 17 line bash script that periodically makes new submission to DeadDrop. In our experiments, this script quickly filled up the journalist interface. This, in effect, created several problems. First, for each submission the journalist has to copy the submitted encrypted file to an external USB drive, transfer it to the Secure Viewing Station, and then decrypt it. Only then will the journalist know that this submission was fake. This type of attack will likely overwhelm journalists by consuming their time. Second, the DeadDrop journalist interface has no method for deleting submissions. This means that an attacker can easily pollute that interface and make it difficult for journalists to use the system. Third, continuous submissions could deplete the entropy pool on the DeadDrop servers and throttle key generation thereby preventing normal DeadDrop operation. Fourth, since Tor provides anonymity by design, it is very difficult to profile an attacker even if he/she comes from a single source. Furthermore, attackers can make clever scripts that make heuristic-based defenses against these attacks difficult. For example, attackers might submit files of random size and at irregular time intervals -- we implemented both of these.

Space Exhaustion

We discovered that it was possible to exhaust the space on the DeadDrop machine by uploading large files. Uploading large files in such a way would hence prevent all future uploads from succeeding. In fact, future uploads would fail silently and neither the journalist nor the source would be any to the wiser (we experimentally verified this issue). We also found that it was possible to exhaust the space on the Secure Viewing Station by uploading a custom zip file of 1MB that when uploaded to DeadDrop and subsequently unzipped on the Secure Viewing Station, would generate files with total space of 1TB. The zip file was designed and handcrafted so that that even if one was to preview the contents of the zip file, it would lie about the size of the unzipped contents. We believe that this attack is of independent interest as it goes beyond, to our knowledge, previous compression attacks, such as the infamous 42.zip file which requires the unzip program to be run with special, recursive flags -- our attack has not such requirements.

Codeword collision

We analyzed the source code and found that a codeword collision is possible. After a code phrase has been generated and used by a particular source, it is possible that a different source (or an attacker) will also be given the same codephrase; we estimated this probability in the Design Analysis section, but verified that it was possible by analyzing the source code. The possibility of collisions means that the second source (or attacker) will be able to snoop on communication between the first source and the journalist.

7.) Deployment at *The New Yorker*

The New Yorker's deployment of DeadDrop is called StrongBox. Though we only have a blackbox view of *The New Yorker* StrongBox deployment, we believe that *The New Yorker* mostly applied a skin to DeadDrop without other major changes. Given this outside view, and not wanting to cause harm through experimentation with the StrongBox system itself, we can make the following observations:

- We believe that *The New Yorker* did not fully run the setup scripts (possibly for the same reasons that we didn't -- see above). We believe this because their Apache server still sends its version information in the server signature HTTP reply header, which would have been disabled by the setup scripts.

- http://www.newyorker.com/strongbox/ should be served over HTTPS to make it more difficult for attackers to launch man-in-the-middle attacks which might modify the page to show a different onion addresses to possible sources. In fact, we suggest that all parts of newyorker.com be served over HTTPS since attackers might use the non-HTTPS parts of newyorker.com to launch attacks on sources. For example, if https://newyorker.com/strongbox/ includes JavaScript from http://newyorker.com/scripts, the attacker could change the JavaScript being served over HTTP in order to modify the content being served over HTTPS. Furthermore, we suggest the use of HSTS for protecting against SSLStrip type of attacks and for browser certificate pinning. We also suggest the use of FRAME-OPTIONS header to prevent against clickjacking.

- The "https://torproject.org" text on the http://newyorker.com/strongbox/ page should be a link in order to lessen the chances that users might mistype the URL and land on a malicious page.

- *The New Yorker* should consider loading the StrongBox instructions in every page on newyorker.com, hide it (via CSS), and include a link in every page (perhaps in the footer of the page) that would cause the StrongBox information to be shown (without causing any additional HTTP requests to the server). This would help to guard potential sources against traffic analysis by giving them plausible deniability.

- Moreover, to reduce the risk of traffic analysis, *The New Yorker* might consider mirroring a copy of the Tor Browser Bundle in order to allow potential sources to download and use the Tor Browser without ever leaving newyorker.com.

- If possible, *The New Yorker* might also consider serving a DVD image of the Tails anonymity VM (https://tails.boum.org). The DVD image should come with instructions on how to boot into and use the Tails VM in order to communicate with *The New Yorker*'s StrongBox.

- Additionally, *The New Yorker* should consider making their website be a Tor entry node -- so that simply connecting does not mean that one is using Tor, nor does it mean that one is browsing the New Yorker and then deciding to submit data to DeadDrop. Before

making *The New Yorker*'s website a Tor entry node, we suggest that *The New Yorker* partner with a security expert as correct deployment requires deep consideration with respect to the packet timing, size, frequency and so on.

- *The New Yorker* should create a GitHub (or other) repository of its deployment. This is necessary so that the community (and any potential source) can verify that *The New Yorker* did not introduce vulnerabilities by modifying the DeadDrop code. For additional security, cryptographic hashes of releases should be published in *The New Yorker* magazines (this doesn't have to be obtrusive and can be hidden from plain view). By publishing release hashes in physical magazines, *The New Yorker* will help to thwart attackers that might compromise the source code repository.

7.2) Test Submissions

We submitted test documents to *The New Yorker*'s StrongBox. As noted earlier, journalists must manually check for submissions, and we wanted to verify that someone at *The New Yorker* was actually checking submissions made to StrongBox. The documents (and messages) in our submission specifically asked for a response. Additionally, to our knowledge, the StrongBox staff was informed about our submissions via a media contact, and was explicitly asked, by that media contact, to look at the submissions and reply. Unfortunately, after more than 9 weeks, we have received no reply – even after several requests.

Additionally, the documents we submitted were specially crafted. Specifically, they were "bugged" to make network requests to our servers with the intention of verifying that StrongBox's staff did indeed use an airgapped machine to check submissions. Unfortunately, we were unable to verify *The New Yorker's* process for checking documents, as they did not reply to our submissions.

8.) Usage Issues

Though DeadDrop core is, for the most part, a technically sound system, we would like to emphasize that there are a number of broader issues that must be considered when trying to provide an anonymous submission site.

8.1) Attestation of Server-side Code

In the current DeadDrop model, sources have no proof that the media organization is indeed running DeadDrop or that it has been installed correctly and securely. In fact, in some cases, the source might not be communicating with the actual media organization as a government might potentially seize the DeadDrop server, issue a "gag" order to the media organization, and continue operating the DeadDrop server in order to catch the source[13].

---

[13] We acknowledge that we know of no such gag order. However, we also don't see any reason why such orders are impossible.

8.2) Anonymity Beyond DeadDrop

Strong attackers might mount attacks that are beyond the scope of DeadDrop, but that might still compromise the user's identity. For example, strong attackers might be monitoring all of the source's network communication, establish video or audio surveillance of their homes or offices, or feed the suspected source "fake" confidential data in order to see if it is leaked. Additionally, documents might be watermarked or specifically altered in ways that will allow adversaries identify the source of the leak. Thus, even if following best practices, the source will be taking a risk by leaking documents.

9.) Conclusion and Recommendations

Based on our evaluation of the DeadDrop design and implementation, we believe that the DeadDrop's core application is a technically decent system for supporting anonymous communication between sources and journalists. However, we are concerned about the level of technical sophistication that journalists are expected to have and that they might, for usability reasons, make mistakes that leak the confidential information about the source. Furthermore, we caution that the system will likely be unable to protect the source against the most powerful type of adversaries which can monitor network flows, confiscate physical machines at will, or watermark documents that the source might try to submit to the journalists. Moreover, anonymous communication over the Internet is incredibly difficult because it can often be breached by one's activities in the physical world. Nevertheless, we do believe that DeadDrop is a step forward in improving anonymous communications between sources and journalists.

If, however, DeadDrop is to be deployed responsibly, we believe that several major issues must be addressed. We list some of the suggestion below and leave the rest in the Appendix; we also discuss recommendations in the earlier sections.

- Develop a set of anonymity best practices and ship these as part of the DeadDrop distribution to be linked to from the landing page.
- Fix the DeadDrop deployment/hardening scripts; we provide great detail on this in the appendix.
- Fix the DeadDrop documentation. Some of it was ambiguous and confusing. We provide specific comments in the Appendix.
- Modify dictionary and remove words with non-ascii characters. Also consider pruning the dictionary to remove very rare words. Though reducing the dictionary size will slightly reduce the effort needed to crack the source's code phrase, we believe the usability tradeoff is worth it and might bring higher security in the end (since the source won't be forced to write the code words down or forget them).
- Use scrypt, PBKDF2, or bcrypt instead of SHA to hash the four codewords. This will make offline attacks much more difficult.
- Consider using ECC instead of RSA for encryption. This will make DOS attacks based on key generation less of an issue.
- We suggest that the documentation suggest the use of "haveged" when a hardware entropy generator is not available.

- Develop and use software versioning best practices (see our detailed discussion in Section 4.2). Deploy packed stable and beta versions of the application along with hashes of the packages.
- Modify the DeadDrop application to display the software version number on every page so that sources can tell whether the latest up-to-date version of the code is being used.
- Consider putting in a CAPTCHA or some other submission thresholding to thwart DOS attacks.
- Develop a set of metadata filters/scrubbers to prevent the source's identity leaking through file metadata. Consider using MAT (Metadata Anonymization Toolkit -- https://mat.boum.org/). Additionally, metadata removal should be used in an intelligent manner as sometimes metadata can serve as evidence. The submission interface should parse the metadata, show it to the source and ask him/her whether this data should be scrubbed or left in tact. Ideally, this should either be done on the source's client so that sensitive metadata never leaves his/her machine. For example, metadata removal could be done by a plugin shipped with the Tor Browser Bundle or anoter program (e.g., MAT is shipped by default in the Tails anonymization VM).
- Develop functionality to allow journalists to delete (or at least hide) files sent by sources from the DeadDrop server.
- Develop functionality to allow journalists to decrypt files on the SVS using a point and click GUI. Journalists should not be required to develop expertise in GPG command line usage.
- The decryption process should render files with an extension equal to the extension with which they were originally submitted. Alternatively, the recommendation for using the "-v" flag (which shows the original filename) should be made more evident in the directions, as we did not find it.
- Develop and document a procedure for loss of journalists' GPG keys -- perhaps consider using Shamir Secret Sharing in order to provide some fault tolerance.
- All relevant cryptographic keys (such as the Tor hidden service keys and journalist keys) should be backed up in a secure way, perhaps using some kind of *k* of *n* splitting scheme (e.g., Shamir's Secret Sharing).
- Prevent access not over Tor by looking for the x-tor2web HTTP header.
- All keydrives used by journalists should be encrypted in order to address secure erase issues on flash memory.

We hope these recommendations help to improve DeadDrop. Finally, we'd like to note that there are a number of related efforts that try to allow anonymous data submission to journalists. Many of these sites are branded with the name "leak" and can be found through directories such as this: http://leakdirectory.org/index.php/Leak_Site_Directory. A comparison of these efforts is beyond the scope of this document.

10.) Acknowledgements

Thank you to Kevin Poulsen for reviewing a draft of this report.

Appendix

A.) Appendix: Documentation Analysis

A detailed assessment of the provided documentation follows.

A.1) THREAT_MODEL.md

The threat model is described in the following document: https://github.com/deaddrop/DeadDropDocs/blob/master/THREAT_MODEL.md. We analyzed the documentation and describe the inconsistencies and ambiguities below. First, we consider the "Anonymity Provided" section:

- "It is highly recommended for the source to use Tor to submit messages, documents and check for replies" -- is this a requirement or just a recommendation? We believe that this should be a technical requirement. That is, it should be impossible to access DeadDrop over a non-Tor connection.
- "Only the two selected journalists have physical access to the application's GPG private key and know the key's passphrase used to decrypt source files. These steps were taken to provide reasonable assurance that only the two selected journalists could decrypt the files after they were encrypted in the application." It is not clear to us which passphrase is being talked about yet.
- "Apache access logs are not kept." -- this statement does not cover other types of logs (such as error logs). The application should strive to keep as few logs as possible since it should be assumed that the machine will likely eventually be compromised, all logs will be extracted and used by attackers.
- "The source's uploaded messages and documents are encrypted before being stored to disk." -- are they encrypted in RAM? The install guide should make sure swap is disabled, but this is not mentioned.
- "The secure viewing station is where the application's GPG private keys decrypts the source's submitted information and is 1) never connected to a network, 2) booted from a LiveCD, 3) the hard drive is removed, 4) physically located in a secured corporate facility." --- what is the process for updating the LiveCD once the image and programs are out of date? How are the submitted files transferred onto it?

Next, we consider the "Application Usage" section:

- The documentation makes multiple references to a *secure keydrive*, but doesn't document what requirements must be met for the keydrive to be considered secure.
- There are a lot of details whose purpose is to provide defense in depth, but which are documented as recommended (implying not required). One example is that encrypted files should be deleted as soon as possible to defend against a situation where a codeword is compromised. It needs to be emphasized which details are necessary or core to the security of DeadDrop, and which are just extra.
- We believe that sources should be adviced to use the Tails LiveCD. This provides better anonymity and is easier to use than the Tor Browser bundle.

- https://github.com/deaddrop/DeadDropDocs/blob/master/THREAT_MODEL.md#authentication documents that a code word is three words together each chosen at random from a word list. The actual implementation, however, uses four words.
- It is not clear what is the correct course of action if the source forgets his/her codewords.
- We found step S5 to be confusing: “The source interface has the rng-tools package installed and configured to use an external random number generator device for the source of entropy in the key generation.” -- it is not clear which interface is being talked about here. How realistic is it to have an external (cryptographically secure) source of entropy? How can a system be provided where both the source and journalist can trust the source of entropy?
- Steps S3 - S6 again raise the question of codewords: how good are the generated codewords? Are they truly easily memorizable? Our experiments in Section 6 indicate that this may not be the case. How well are the codewords chosen? From the documentation, it’s not clear if DeadDrop remember which codewords were chosen or if they can be re-chosen so two different sources collide on a mailbox (albeit with low probability)? Our implementation inspection in Section 6 show that a collision is possible and is not handled properly.
- Step S8 mentions a “background lambda process”. We are unsure what that is. After looking at the documentation, we believe that the authors are referring to the use of key generation through the use of a background thread. This line should be reworded or removed from the documentation.

Next, we consider the “Journalist’s Role” section:

- Presumably there should be a (J0) where the Journalist’s TLS Client Cert is securely generated?
- “The source code IDs that are presented to the journalist is a different 3-word code IDs from the source’s clear text codename. The application generates the separate code IDs using the hashes of the sources’ codenames. This is done so that the source's clear text codename is not known to the journalist.” -- given our analysis, it would not be infeasible to brute force the hashes.
- “(J7) For scenarios where a journalist requires part of the unencrypted contents of submitted information for publication, the journalist should encrypt the clear text contents using their personal GPG keypair -- not the application's keypair, on the secure viewing station before transferring them to their personal workstation. The source's documents and messages should be encrypted at rest until the article is ready for publication.” -- we foresee this being difficult for journalists to do. The presence of multiple keys could lead to confusion about which key to use. Additionally, it’s not clear how the journalists should get their personal GPG keys to the secure viewing station without plugging an insecure flash drive into the secure viewing station.
- “(J8) The journalist's interface also has a reply function. The journalist can enter their message for a specific source into a text box and click 'Submit.' The application retrieves the source's GPG public key based off of the source's hashed codename. If the journalist cannot access the source's GPG public key, the reply function is not rendered.

The application encrypts the journalist reply with a unique source's GPG public key, which is stored in the source's hashed codename-encrypted file store." -- it's not clear when the source's GPG public key might not be accessible to the application.

- (J2): "The source code IDs that are presented to the journalist is a different 3-word code IDs from the source's clear text codename". -- the structure/format of the codename hasn't been discussed prior in this document, so it's hard to understand what this means.
- (J6): "After the transfer is complete, the journalist should securely delete the encrypted file from the external hard drive and remove the drive". -- the process for securely deleting files should be stated.

The "Authentication" and "Security Monitoring" sections are loaded with technical jargon and may seem somewhat overwhelming. Additionally, the following quote is unclear as the "system.random" doesn't correspond to any code that we could find: "The codename is derived by using "system.random" to randomly choose 3 words from the pre-defined word list file. It is supplied by the source in each request to a protected resource to authenticate."

A.2) README.md (Installation Intructions)

https://github.com/deaddrop/deaddropdocs/
This section should start with something like:

- You will need WWW computers
- You will need ZZZ USB keys
- You will need YYY hours of time
- You will need YYY CDs/USBs with ISOs of XXX

Otherwise, it's not clear what one needs in order to begin.

Misc

- The diagram mentions an iLo switch at the bottom, connected to source, journalist, monitor. Does that mean the system depends on HP hardware? Does iLo allow full system control? Dan Farmer's work (http://fish2.com/ipmi/itrain.html) indicates that use of such technology may not be the best idea.
- The guide offers no checkpoints as ways to verify if previous steps went well
- Copy (cp) commands should standardize use of trailing-slash for clarity
- The instructions say to prevent split-tunnel VPNs. To do this, we believe one has to use proprietary VPN software (e.g., Cisco's).

Local CA Install

- The documentation discusses that the box should not be connected to a network, yet it describes the use of apt-get to update the machine (in the "Setup OpenSSL" step).
- Setup instructions mention that we want to use LUKS, but the Ubuntu install process doesn't mention that term by name (maybe encrypted LVM)
- Is the CA supposed to be an Ubuntu server or desktop? The directions don't specify.

Local CA Install Notes

- The Local CA machine is not supposed to have internet connectivity, yet it describes use of *apt-get update* and *apt-get upgrade*.
- Instructions don't mention becoming root or non-root user, but:
  - Use of sudo is described for for *apt-get* (implying non root)
  - It is necessary to make a directory in *opt* (implying root privileges)
- Modifying /usr/lib/ssl/openssl.cnf
  - It should be clarified to modify only the line containing *"dir ="*
- Received error when running first openssl
  - "using curve name prime256v1 instead of secp256r1".
- To follow pem conventions, the journalist interface key should be named journalist.without.key.pem instead of journalist.without.key.
- 'Copy the needed certs to the app's external hard drive' section is unclear or incorrect in various ways.
  - The first command misspells the directory name.
  - second command says .wo instead of .without.
  - second/third/fourth commands assume some directory's existence.
  - Commands tell you to move to a directory that isn't an external hard drive.

Configure Secure-Viewing-Station and Application's GPG keypair

- Ubuntu LiveCD's can be created with reserved extra storage. A LiveCD with this feature enabled allows programs to write back to the flash drive containing the ubuntu install. This means that a program can violate the volatility of the Secure Viewing Station by permanently installing other programs to it.
- The guide describes dependencies to be downloaded through *apt-get*, yet the secure viewing station must never be given a network connection.
- The dependencies listed actually cannot be immediately downloaded onto a machine running off of the live CD because they come from the universe repositories, which are not included on the live CD (but do show up after an install).
- The command to generate the keypair doesn't give a recommended expiration time to configure for the keypair.
- The command to export the public key is missing the homedir option
- The command to export the public key uses the extension acs instead of asc
- The command to dump the gpg key fingerprint has an incorrectly prewritten homedir.
- We didn't get the Command> prompt like it describes
- This stage took about 30 minutes with all of the fumbling on incorrect instructions

Install and Configure Puppet

- All of the files in puppet are inexplicably labeled executable.
- It's not explained why iptables-persistent is installed before puppet. That can be installed using puppet.
- Copying of puppet modules into /etc/puppet/modules should have the recursive flag set.

- The documentation should describe that copied file permissions need to be set readable similarly to files around them.
- Instructions for puppet.conf don't describe exactly where in the file to add the cited configuration options.
- For puppet.conf, *dbadapter* is misspelled as *dbadpter*.
- "Gather the required files from the external hard drives"
    - Which is "App's pub gpg key", etc? These should be explicit filenames.
    - We believe that the application's public gpg key should be *journalist.acs* (as opposed to the documented *.asc*)
- The nodes.pp configuration file has multiple sections that need modification. The document only mentions modifying the first section and does not mention that the hostnames in the latter section needs hostnames to be added.
- In the puppet modules, there were various important sections of code commented out. When adding these sections of code back, we found that they either didn't work or depended on other modules that were themselves incomplete in some way. It is clear that these modules were never completed.
    - Line 19 of source.pp, *include deaddrop:apparmor*, is commented out. Without it, an error related to the AAHatName directive is given when starting apache.
    - After running the puppet agent on journalist and source servers the screensaver (gnome-screensaver) no longer would accept our password. Errors in log report: "unix_chkpwd: [] check pass; user unknown. gnome-screensaver-dialog: pam_unix(gnome-screensaver:auth) authentication failure."
    - vcsrepo (line 60-66) in journalist.pp is commented out. This prevents the files from the root of the git repo from getting downloaded.
- We ended up manually cloning the deaddrop.git repository and dumping all files inside of it into the */var/www/deaddrop* directory (along with the files that were generated by puppet). The *static* file generated by puppet was removed and replaced with the static directory from the deaddrop repository. We did the same for the source machine.
- The apache configuration as given can't correctly run the source web server. It embeds the python application inside of an apache thread, yet the application tries to fire off a separate thread to generate gpg keys, which fails. Instead, we have used WSGI daemon mode in order to get the application to launch separate threads.
- Puppet didn't correctly install an ssh server on the journalist or the source, nor does it preconfigure sshfs. Furthermore, there is no instruction on the parameters of how sshfs should be set up. What cipher, key lengths, and options should be used? The choice can either make the setup reasonably secure or insecure.
- None of the documentation told us exactly what state our machines should end at. This made it difficult to reverse engineer the intention of the puppet scripts