Report of Security Audit of *OONI*

- 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

The Open Observatory of Network Interference, *OONI*, is an open source network testing framework and associated tests for detecting internet censorship. This report is the result of a security audit by the Least Authority consultancy, paid for by the Open Technology Fund.

About Us

Least Authority was founded to provide open source cloud storage technology that preserves user privacy and control over their data. Our expertise lies at the intersection of open source software, cryptographic applications, decentralized networking, and user-empowering technologies.

We also provide consulting services for clients who share our vision of open, transparent, secure technologies which protect and empower users. To that end we provide security auditing services, focusing on transparent processes for open source projects.

Schedule

The audit was performed from 2014-03-17 through 2014-04-04. This report is the final draft of the audit findings, delivered on 2014-04-17.

Audit Scope

This audit primarily focused on the ooni-backend software component, although we also reviewed ooni-probe where time allowed. We focused on standard functional and operational application security.

This audit explicitly excluded considerations of legal, professional, and ethical liability of data collected from *OONI* tests. Existing considerations of such risks are available in the OONI Threat Model, specifically the non-resource-risk documented threats.

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, 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 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 *OONI*, 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

Target Code

Revision

This audit targets the ooni-backend v1.0.0 and ooni-probe v1.0.1 release revisions, which comprise the server and client components of *OONI*.

Dependencies

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

Both *ooni-backend* and *ooni-probe* depend on:

PyYaml
: A YAML parser and emitter.

Twisted
: An asynchronous I/O and scheduling framework.

ipaddr
: A library for working with IPv4 and IPv6 addresses.

txtorcon
: A *Tor* controller client.

pyOpenSSL
: A wrapper for the *OpenSSL* library.

ooni-backend also depends on:

cyclone
: A web application framework for *Twisted*.

pygeoip
: A library to look up geo-locations of IP addresses.

transaction
: A transaction management library.

zope.component*, *zope.interface
: An abstraction framework for Python, used by *Twisted*.

zope.event
: An event publish-and-subscribe system.

pysqlite
: An interface to *SQLite*.

ooni-probe also depends on:

geoip
: An apparently *different* library to look up geo-locations of IP addresses. (It's in `requirements.txt` without a version-number constraint, too.)

txsocksx
: SOCKS client endpoints for *Twisted*.

parsley
: A library to parse with parsing-expression grammars.

dnspython

A DNS toolkit for Python.

scapy-real

A packet-manipulation library.

pypcap

A wrapper for the packet-capture library *libpcap*.

Target Configuration

We tested *ooni-probe* and *ooni-backend* with the default configuration in `oonib.conf.example` and `ooniprobe.conf.sample` respectively. We focused mainly on the `mlab.deck` test deck.

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 *OONI* 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 by successfully exploiting the vulnerability. We make this assertion conservatively, and do not include a "real life impact analysis" such as determining how many existing users could be compromised by a live attack.

Preconditions: These are conditions necessary for the vulnerability to be exploitable. For example, if the vulnerability only exists for a certain configuration, it will be noted here.

Feasibility: The feasibility of an attack is a *rough* and *tentative* educated guess as to how difficult it is to perform the attack, assuming the preconditions are met.

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* turns out to be unexploitable, the verification section becomes especially important, because mistakes in verification may mask exploitability.

Technical Details: In this section we describe the implementation details and the specific process necessary to leverage 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. CSRF Token Not Compared in Constant Time

Reported: 2014-04-10

Applies To: *ooni-probe*

Synopsis:

The CSRF token is not compared in constant time. It may be possible to extract CSRF tokens through a side channel attack to make cross-domain requests.

Impact:

An attacker, for example a malicious web page that the user visits, may be able to perform Cross-Site Request Forgery attacks after learning the CSRF token. These attacks could result in changes to *ooni-probe's* configuration, or could result in tests running without the user's consent.

Preconditions:

The attacker must be able to make requests from the user's browser to the *ooni-probe* server and time the response.

Feasibility:

The number of requests the attacker has to measure depends on the specific implementation of Python's `!=` operator, which we did not investigate.

Verification:

This issue was verified by inspecting the source code. We did not create a proof-of-concept exploit for this issue.

Technical Details:

The following code is used to check the CSRF token. It can be found in `ooni-probe/ooni/api/spec.py`.

```
def check_xsrf(method):
    @functools.wraps(method)
    def wrapper(self, *args, **kw):
        xsrf_header = self.request.headers.get("X-XSRF-TOKEN")
        if self.xsrf_token != xsrf_header:
            raise web.HTTPError(403, "Invalid XSRF token.")
        return method(self, *args, **kw)
    return wrapper
```

The `!=` operator compares strings byte by byte (or word by word), and stops on the first difference. This small timing difference is usually enough to extract the string being compared against by making repeated requests. It is sometimes possible to do this timing measurement cross-domain.

Mitigation:

Current *ooni-probe* users can mitigate this issue by turning off or disabling the web administration pages.

Remediation:

Use a well-vetted constant-time comparator to check the CSRF token.

We recommend using the `time_safe_compare` method from Tahoe-LAFS:

```
def timing_safe_compare(a, b):
    n = os.urandom(32)
    return bool(tagged_hash(n, a) == tagged_hash(n, b))
```

Status: Reported.

Issue B. Arbitrary File Write in Input File Uploader

Reported: 2014-04-10

Applies To: *ooni-probe*

Synopsis:

The *ooni-probe* web interface has a page for uploading test inputs. The server-side logic that handles file uploads is vulnerable, as it allows the uploaded file to be written anywhere in the filesystem.

Impact:

An attacker can create a new file with arbitrary contents anywhere on the *ooni-probe* server's filesystem. Usually it's possible to execute code remotely this way, e.g. by writing a script into `/etc/cron.daily/`.

Preconditions:

The *ooni-probe* web interface must be exposed to the attacker, or the attacker must be able to fool or force (e.g. by CSRF) a legitimate user into making the file upload request.

Feasibility:

If the attacker has access to the web interface, the attack is very easy to perform and can be automated.

Verification:

This issue was verified by inspecting the code, and also by using the *TamperData* extension for Firefox to upload a file with the name `test/../../../../../../tmp/test.txt` which successfully created a `/tmp/test.txt` file on the *ooni-probe* server.

When we verified this issue, the CSRF tokens were not working properly, so we had to modify the code to remove the `@check_xsrf`. If the CSRF token system was working properly, its presence would not have stopped the attack.

Technical Details:

The code that handles the file upload is in `ooni-probe/ooni/api/spec.py`. It is reproduced below.

```
@check_xsrf
def post(self):
    """
    Add a new input to the currently installed inputs.
    """
    input_file = self.request.files.get("file")[0]
    filename = input_file['filename']

    if not filename or not re.match('(\w.*\.\w.*).*', filename):
        raise InvalidInputFilename

    if os.path.exists(filename):
        raise FilenameExists

    content_type = input_file["content_type"]
    body = input_file["body"]

    fn = os.path.join(config.inputs_directory, filename)
    with open(os.path.abspath(fn), "w") as fp:
        fp.write(body)
```

The filename is checked against a regular expression. However, the regular expression matches filenames that contain any number of `../` sequences. An attacker can upload a file with these characters in the name, and the file will be written outside of the configured `inputs_directory`.

Mitigation:

Current *ooni-probe* users should disable *ooni-probe's* web interface, or ensure that only trusted people can make requests to it.

Remediation:

Use `twisted.python.filepath.FilePath`.

Status: Reported.

Issue C. User Input Written to Logs

Reported: 2014-04-10

Applies To: *ooni-probe* and *ooni-backend*

Synopsis:

User input is written to the log files without escaping/removing characters with special meaning. An attacker can insert fake log entries or create log entries that contain terminal escape codes, or other injection attacks.

Impact:

An attacker is able to create fake log entries which could confuse the system administrator or hide an attack. An attacker can also inject terminal escape sequences, buffer overflow attacks, or other malicious byte sequences against a variety of system administration tools, so this vulnerability opens up a large attack surface.

Preconditions:

Logging must be enabled and set to a log level that encompasses log messages containing user input (this is the default).

Feasibility:

The attack is possible if the attacker can make requests to the *oonib* or *ooni-probe* servers, or if the attacker can provide any other kind of input that ends up in the logs.

Verification:

This issue has been verified by inspecting the code.

Technical Details:

For a concrete example, see `oonib/testhelpers/http_helpers.py`, where malformed headers are written to the logs:

```
def headerReceived(self, line):
    try:
        header, data = line.split(':', 1)
        self.headers.append((header, data.strip()))
    except:
        log.err("Got malformed HTTP Header request field")
        log.err("%s" % line)
```

Note: The above code actually does not work, because `log` is undefined.

Another example exists in *ooni-probe*, in `ooni/geoip.py`. In the following code, the `self.address` variable comes from the network without being sanitized (i.e. through the `UbuntuGeoIP` class).

```
try:
    yield self.askGeoIPService()
    log.msg("Found your IP via a GeoIP service: %s" % self.address)
    defer.returnValue(self.address)
except Exception, e:
    log.msg("Unable to lookup the probe IP via GeoIPService")
    raise e
```

Mitigation:

Users running either *oonib* or *ooni-probe* should be aware that log entries can be spoofed. Users should also view the log with a text editor rather than their terminal to avoid terminal escape attacks. Note that a

text editor also may have buffer overflow vulnerabilities, so we recommend ensuring the text editor is updated frequently for security patches.

Remediation:

Rather than escape or sanitize user input at each call to `log()`, we recommend fixing the log function itself to do this. 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:

```
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: Reported.

Issue D. Tor Build Script Downloads zlib Over HTTP

Reported: 2014-04-10

Applies To: *ooni-backend*

Synopsis:

The Tor build script downloads the *zlib* code over an insecure connection.

Impact:

The attacker can potentially execute arbitrary code on the user's system.

Preconditions:

This issue is only exploitable if the *oonib* administrator manually runs the `build_tor2web_tor.sh` script.

Feasibility:

To exploit this issue, the attacker must intercept the *zlib* code as it is being downloaded and replace it with a malicious copy.

Verification:

This issue was verified by reading the `build_tor2web_tor.sh` script.

Technical Details:

The `build_tor2web_tor.sh` script downloads code to be built from the following URLs:

```
# Package URLS
URLS="\
https://www.torproject.org/dist/tor-$TOR_VERSION.tar.gz
https://www.torproject.org/dist/tor-$TOR_VERSION.tar.gz.asc
http://zlib.net/zlib-$ZLIB_VERSION.tar.gz
https://www.openssl.org/source/openssl-$OPENSSL_VERSION.tar.gz.asc
https://www.openssl.org/source/openssl-$OPENSSL_VERSION.tar.gz
https://github.com/downloads/libevent/libevent/libevent-$LIBEVENT_VERSION.tar.gz.asc
https://github.com/downloads/libevent/libevent/libevent-$LIBEVENT_VERSION.tar.gz"
```

All URLs are HTTPS except for the *zlib*, which is HTTP. An attacker could intercept the *zlib* download connection and inject malicious code, so that Tor is built with a malicious copy of *zlib*.

Mitigation:

OONI users can protect themselves by not using this script, and installing these dependencies with a secure package distribution mechanism such as *apt-get* on *debian*.

Remediation:

The integrity of the *zlib* code should be verified. This could be done in one of the following ways:

- Include a hard-coded SHA256 checksum of the file, which gets checked after it has been downloaded.
- Host a copy of *zlib* on a server that supports secure connections.
- Encourage the *zlib* project to support HTTPS and/or GPG signatures.

Status: Reported.

Additionally, we verified that *curl* attempts to verify certificates against a trust root. On our *debian* system, we observed it used the `/etc/ssl/certs` directory for this purpose.

Issue E. Denial of Service by Uploading Lots of Header Lines

Reported: 2014-04-10

Applies To: *ooni-backend*

Synopsis:

By uploading megabytes of headers an attacker can consume the server's CPU.

Impact:

An attacker can deny service to others by using relatively modest bandwidth.

Preconditions:

A collector using `SimpleHTTPChannel` must be accessible.

Feasibility:

The attacker must send a few megabytes of short lines for the headers of an HTTP request, and keep the connection open.

Verification:

This issue has been verified by inspecting the code.

We did not confirm the vulnerability on a running *ooni-backend* instance, only on a unit test of the most relevant line of code.

Technical Details:

In `oonib/testhelpers/http_helpers.py` the `SimpleHTTPChannel` has unbounded buffers `self.headers` (the list of headers) and `self.__header` (the latest header line). *Twisted* limits the line lengths, but the latest header can be extended indefinitely:

```
elif line[0] in ' \t':
    # This is to support header field value folding over multiple lines
    # as specified by rfc2616.
    self.__header = self.__header+'\n'+line
```

Each extension line takes time linear in `len(self.__header)`, adding up to time superlinear in the number of lines. In an experiment on a laptop with just this line of code isolated, after a megabyte of `' \n'` repeated as input it was taking about 0.1 milliseconds of CPU per extra byte of input.

Mitigation:

Current users of *ooni-backend* could disable HTTP helpers, or run them behind a proxy that guards against overlong extended headers.

Remediation:

Change the code to bound the length of `self.__header` and to enforce the currently-unused `maxHeaders` limit on `self.headers`. Also, use `+=` or `string.join` to concatenate strings, to take linear instead of superlinear time overall.

Status: Reported.

Issue F. oonid Lacks Authentication Checks

Reported: 2014-04-10

Applies To: *ooni-probe*

Synopsis:

By default, *oonid* listens on a public IP address and does not have a mechanism for authentication. This allows anyone who can connect to the daemon to run tests, which may aid malicious attacks, such as using the TCP connection test for port scanning.

Impact:

The attacker can coerce the *ooni-probe* daemon into performing attacks against other systems without the operator's consent.

Preconditions:

ooni-probe must be installed and *oonid* must be running and listening on a public IP address.

Feasibility:

Exploitation is easy, since it can be done using a web browser to connect to the *ooni-probe* daemon, upload test inputs, and run tests. This can also be automated with a script that makes requests to the API.

Verification:

This vulnerability has been verified by source code inspection and by running the *oonid* in the GitHub repository.

Technical Details:

The TCP server is created in `ooni-probe/ooni/oonid.py`:

```
def getOonid():
    director = Director()
    director.start()
    oonidApplication.director = director
    return internet.TCPServer(int(config.advanced.oonid_api_port), oonidApplication)
```

Mitigation:

Users can mitigate this risk by setting up firewall rules to prevent unauthorized access to *oonid*.

Remediation:

An authentication mechanism should be added to *oonid*.

Listening on `127.0.0.1` may not completely eliminate this problem, since unprivileged code (running as a regular user) on the same system can still use the API.

Status: Reported.

Issue G. Cross-Site Scripting in HTTPRandomPage

Reported: 2014-04-10

Applies To: *ooni-backend*

Synopsis:

The `HTTPRandomPage` helper reflects user input in the output without escaping it. This helper is currently disabled so it does not pose an immediate security risk.

Impact:

An attacker can execute arbitrary scripts in the same origin as the HTTP helper.

According to the Google Browser Security Handbook, cookie scope is not bound to the port number, so a script running in this context might be able to read and write cookies of other websites on the same domain.

Preconditions:

The `HTTPRandomPage` helper is currently disabled, so it cannot be exploited.

Feasibility:

The attacker must get the user to click a malicious link.

Verification:

This issue has been verified by inspecting the source code.

Technical Details:

The following code appears in `testhelpers/http_helpers.py`. The `all()` method handles all requests (GET, POST, etc.):

```
def genrandompage(self, length=100, keyword=none):
    data = self._gen_random_string(length/2)
    if keyword:
        data += keyword
    data += self._gen_random_string(length - length/2)
    data += '\n'
    return data

def all(self, length, keyword):
    length = 100
    if length > 100000:
        length = 100000
    return self.genrandompage(length, keyword)
```

Mitigation:

Current users are not at risk because the `HTTPRandomPage` helper is disabled.

Remediation:

To remediate this issue, either remove the `HTTPRandomPage` code or make it safe. To make it safe, escape the reflected input or set the `Content-Disposition` header so that browsers will download the file instead of interpreting it as HTML.

Status: Reported.

Issue H. nettest_to_path Does Not Sanitize the NetTest Name

Reported: 2014-04-10

Applies To: *ooni-probe*

Synopsis:

The path to the Python script containing the test implementation is constructed in an unsafe manner.

Impact:

An attacker that can provide a test deck file can execute arbitrary code.

Preconditions:

The user must use a test deck provided by the attacker.

Feasibility:

Exploiting this issue is easy if the attacker can provide a malicious test deck file and can write a Python script anywhere in the filesystem (e.g. in `/tmp/`).

Verification:

Verified by code inspection.

Technical Details:

In `ooni-probe/ooni/deck.py`, the `nettest_to_path()` method constructs the path to the NetTest Python script from the NetTest name as follows:

```
def nettest_to_path(path):
    """
    Takes as input either a path or a nettest name.

    Returns:

        full path to the nettest file.
    """
    path_via_name = os.path.join(config.nettest_directory, path + '.py')
    if os.path.exists(path):
        return path
    elif os.path.exists(path_via_name):
        return path_via_name
    else:
        raise e.NetTestNotFound(path)
```

This is unsafe, because if the attacker can control the `path` parameter, they can set it to something like `foo/../../../../tmp/evilscript.py` to execute arbitrary Python code with the same permissions as *ooni-probe*.

Mitigation:

Current users can mitigate this risk by only using test decks from trusted sources or manually verifying the `test_file` parameter of the test deck.

Remediation:

Use `twisted.python.filepath.FilePath`.

Status: Reported.

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

- Cryptographic signatures of downloaded software are checked in the `build_tor2web_tor.sh` script. This is a very good practice.

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

- Don't rely on regular expressions. There are several cases where regular expressions are relied on for sanitizing or checking data. This is usually a bad thing, since it's so easy to get regular expressions wrong.
- If you do use regular expressions, use raw string literals. That is, `r'foo'`, not `'foo'`. All of the regexes in *ooni-backend*, outside the API routers, are non-raw strings which happen to work even though most include backslashes, but it's a mistake waiting to happen.
- There's another problem with the regular-expression checks for security. Consider for example `report_id_regexp = re.compile("[a-zA-Z0-9_\-]+$")`, the regexp for the reports that include the timestamp. The Python documentation says that in non-`MULTILINE` mode `$` will match the end of the string *or just before the last newline in the string*.

  So these regular expressions allow a newline at the end of the report id, and various other things that go into file paths and such. Is that exploitable? We did not investigate in the time allotted.

  As a patch for this particular issue with regular expressions (insofar as they are kept), we suggest removing the `$` and calling the following `match_exactly` in place of `re.match` (with the proviso that it's only lightly tested):

```
import re

def match_exactly(pattern, string, flags=0):
    """Return an re.match object if `pattern` matches the whole of
    `string`, else None. `pattern` should be a regex in compiled or
    string form.

    (It's conventional to do this by calling re.match with a pattern
    ending in $ -- but that would allow an optional newline at the end
    of the string, regardless of the pattern.)
    """
    m = re.match(pattern, string, flags)
    return m if m and m.end() == len(string) else None
```

- Don't include external input into filenames. It's hard to sanitize strings for file names, since the precise rules depend which operating system you're running on. It's better to not put user-input strings in filenames at all. Making this change would save a lot of auditing effort.
- If you do use input in filenames, instead of `os.path.join` use `twisted.python.filepath.FilePath`, a file-path abstraction designed with the possibility of attack in mind.
- In general, sanitize or check variables holding user input systematically in a way that makes safety *locally* obvious.

  For an example of going against this principle, the `NewReportHandlerFile#post()` method appears to be vulnerable because the `probe_asn` variable is used to create the report filename. It's in fact not vulnerable, because the value of `probe_asn` is checked by a regular expression earlier on in another method.

  For another, this note appears in `oonib/deck/handlers.py`: "we don't have to sanitize deckID, because it's already checked against matching a certain pattern in the handler."

  The preferred way to ensure locally-obvious safety is by converting any dangerous input at the boundary of the system immediately into a type that's not directly substitutable for strings. (`FilePath`, above, is an example of such a type for paths.) If you still want checks that don't fit this pattern, it's better (less work to audit and maintain) to place them directly before the danger spot.
- Set up a security contact email address. The *OONI* project should publish an email address and public keys so users can report vulnerabilities securely.
- In *ooni-probe*, there is a dangerous method `loadNetTestString()` that allows remote code execution if the parameters are malicious. It appears to be unused, so it should be removed. If not removed, then it should be made safe so that even if the parameters are malicious, code execution is not possible.
- To better avoid Denial of Service attacks, set `PYTHONHASHSEED=random` so that hash tables are randomized. This can be done with the `-R` command-line option.
- Run PyFlakes regularly. It would have found the undefined-`log` bug, and all its other suggestions for *ooni-backend* are worth following as well. It points out many more probable bugs in *ooni-probe*, which we did not investigate for this report.
- Eschew `import *` because it makes code fragile and harder to audit. For example, *PyFlakes* loses much of its bug-finding power in the presence of `import *`. Where a shorthand is needed, define an abbreviation like `import foo as F`.
- The code commendably uses `yaml.safe_load` instead of `yaml.load`, but in some cases uses `yaml.dump` instead of `yaml.safe_dump`. This seems unnecessary.
- Remove unused dependencies. From reading the code it appears that *ooni-backend* doesn't use *transaction* or *pysqlite*.
- *ooni-backend* uses `randomStr()` to generate report IDs. It is not cryptographically secure, which means report IDs are guessable and more likely to collide. If that is a problem, then a cryptographically secure random number generator should be used to create the report ID.
- The `timestamp` method in `ooni-backend/oonib/otime.py` is susceptible to the "midnight is false" bug which may lead to incorrect dates being used.
- Be more specific about catching exceptions. For example, in `ooni-backend/oonib/otime.py`, there is an `except:`, which will catch all exceptions, possibly ones that it was not intended to catch.
- Use `//` for integer division since `/` will produce floating point numbers in Python 3. `/` is used for integer division in `ooni-backend/oonib/testhelpers/http_helpers.py` in the `genRandomPage` method.
- Avoid possible TOCTTOU bugs like the following from `ooni-backend/oonib/report/handlers.py`. Instead, catch the exception and check the errno.

```
if not os.path.isdir(dst_path):
    os.mkdir(dst_path)
```

Future Work

Unsanitized Input in File Paths

We did not have time to examine every case where file names are constructed from input strings. Some appear to be not exploitable because of checks elsewhere, but we cannot be certain about this. We recommend that all paths constructed from input be considered vulnerable until explicit sanitization (or other defenses) are added to each one.

An incomplete list of the file paths we did not have time to check follows:

- The constructor of the `InputFile` class in `ooni-probe/ooni/deck.py`.
- The `cached_file` method of the `Deck` class in `ooni-probe/ooni/deck.py`.
- The `generatePcapFilename` method in `ooni-probe/ooni/settings.py`.
- **In `ooni-backend/oonib/report/handlers.py`:**
    - `get_report_path`
    - `Report#close`
    - `UpdateReportMixin#updateReport` (Note: Even with proper sanitization, this lets an attacker overwrite *any* existing report if they know the id).
- A bunch of `os.path.join` calls in `ooni-backend/bin/archive_oonib_reports`. This was known to be vulnerable before GitHub Pull Request 40. Currently the *testName* variable is not validated, however it is validated in `report/handlers.py` before being written to the file.

OONI Backend Lacks Authentication

ooni-backend lacks an authentication mechanism. It seems to allow anyone to access all of the information it has such as the test decks and test inputs. Is this a risk?

ooni-backend does, by default, listen on `127.0.0.1:80` which restricts access to users on the same machine. However, this may not be enough if the attacker has access to an unprivileged user account on the system.

Open Questions & Concerns

- Can the regular-expression newline issue, described in Coding Practices above, be exploited?
- *OONI* creates and writes files in home directories. Sometimes this is a different home directory from the user it is running as (e.g. when it's running under *sudo*). Does this create vulnerabilities analogous to `/tmp/` file vulnerabilities, e.g. symlink attacks, etc.?
- We did not audit any of OONI's dependencies. Some of them may benefit from an audit.
- Do the *ooni-backend* test helpers allow reflected attacks? For example, can the DNS helper be used to perform a reflected Denial of Service attack?
- What are the anonymity requirements for *ooni-backend*? The setup instructions mention server-side anonymity. However, we did not analyze attacks that could compromise the *ooni-backend* server's anonymity.
- Why are non-blocking file descriptors used in `ooni-backend/oonib/report/handlers.py`? Note that code here is not properly checking for error conditions during the write. The `except` block will only catch errors during file open, since `writeToFD` indicates error status by its return value.
- The NetTestCase class overrides repr so that user input is included directly in the result. Where is this used?

- The timeout for the HTTP Return JSON Headers is 12 hours. This could make Denial of Service attacks easier.
- The checking of `txtorcon_version` in in `ooni-backend/oonib/runner.py` may be incorrect. It checks that `txtorcon_version < '0.9.0'`. This would be false for the version number `'0.10.0`, if it existed.
- It looks like `validateNettest` in `ooni-backend/oonib/policy/handlers.py` might not be doing what it is supposed to, since it passes whenever `self.nettest` is nonempty and does not check the actual contents.
- The `validate_report_header` method in `report/handlers.py` validates the required fields, but does not check if there are extra fields. Could extra fields in the report be dangerous?
- In `bouncer/handlers.py`, the following code looks up a value in a `dict()` just before checking if the `dict()` contains the key. This is backwards, since an exception will be thrown before the explicit check fails.
- In `report/handlers.py`, the `parseNewReportRequest` method checks the `str()` of the fields, but returns the parsed JSON, leaving code elsewhere responsible for running `str()` a second time on things which might not be strings. It should be made so that parsing converts the fields once and for all.