Thesis Proposal:
Python Static Code Security Analysis

Jurriaan Bremer
6297196
jurriaanbremer@gmail.com

February 8, 2013

1 Introduction

Through this proposal I will present an, in my opinion, very interesting project on which I'd like to do my thesis.

1.1 Motivation

Besides studying I spend most some of my free time studying and researching computer security related issues as I find these very interesting. For example, I'm one of the main developers of Cuckoo Sandbox, which is an automated malware analysis system. Other than that, I've got experience with various topics, such as Reverse Engineering, analyzing custom and/or proprietary (network) protocols, language safety in general (think locating security issues and securing software), exploitation of these bugs, etc.

Given my interest in computer security, I think this project is a very good match as project for my thesis.

1.2 Cuckoo Sandbox

Cuckoo Sandbox is a well-respected project in the security world and is internationally used by people and companies ranging from hobbyists to (big) companies to government agencies.

Cuckoo monitors potential malware by dynamically logging its behavior. It does this by executing the malware inside a virtual machine. My involvement in this project is all the low-level stuff, i.e., I develop the component that actually analyzes and logs the sample [1], as well as the communication between the component and Cuckoo.

Furthermore, I brainstorm and suggest improvements on other parts of the code, e.g., improvements on the design of Cuckoo, performance issues, new or extended functionality, etc.

[1] a sample is any kind of potential malware, an office word file, a pdf file, etc.

2 The Project

The official page for the project can be found as Bug 811876 [2] on the Mozilla website. (Note that, although the link doesn't mention my name *yet*, I've already been accepted as participant, due to my past experience, as partially described above.)

2.1 Goals

The goal of this project is to automatically identify and report common mistakes, vulnerabilities, and other security flaws in applications using Python web-frameworks as basis. As of now I couldn't find any known tools with similar capabilities; there are some static code analysers, but not with security in mind (e.g., syntax highlighting.)

2.2 Interesting Applications

The scope of this project is mostly limited to analyzing and detecting flaws in websites using Python web-frameworks.

2.3 Vulnerabilities

For a brief list of possible vulnerabilities, see also the following link [3].

2.4 Scientific Research

Given the fact that Python is a dynamically typed language, there will be quite some challenges to overcome when it comes to analyzing the source code.

2.4.1 Research Contributions

- Static Taint-Analysis for Python, a Dynamically Typed Language
- A set of Security Rules, describing possible flaws
- Benchmarks for testing the Security Rules

2.4.2 Python Code Analysis

In order to cope with the analysis of Python, a dynamically typed language, we will first get the Abstract Syntax Tree of each invididual source file, which can be retrieved using the built-in *ast* module. We will then translate all code to be in Single Static Assignment (SSA), an intermediate representation in which every single variable is assigned only once. [1]

From there on we will perform Taint Analysis in order to find *Sources*, *Sinks*, and *Sanitizers.* Sources, Sinks, and Sanitizers are based on the set of rules, as referenced in the subsection *Research Contributions.*

We will process the various functions using a *flow-insensitive* algorithm, e.g., take the following code.

[2] `https://bugzil.la/811876`

[3] `https://wiki.mozilla.org/Python_Static_Code_Analysis#Practical_Goals`

```
class A:
    def a(self, value):
        self.abc = value

    def b(self, value):
        self.def = value
```

In this sample code, we cannot accurately, in most cases, determine whether function *a* will be called first or function *b*. Therefore we assume that, when function *a* is called, the member *def* already exists (which is only defined when function *b* is called.) This results in the fact that when function *b* is called somewhere in the code with source data, i.e., data we don't want to emit into the sink directly, then the *def* member will be tainted as well. This is called *flow-insensitive* analysis.

However, when analyzing single functions, such as function *a*, we use a *flow-sensitive* algorithm, which is automatically the case since we use an intermediate SSA representation, as described earlier in this subsection. Take for example the following code.

```
class A:
    def a(self, value):
        a = 42
        # do something
        # ...
        self.abc = value
        return self.abc
```

In this code, the return value *self.abc* is only known after it is assigned with *value*. In other words, when the local variable *a* is assigned, the *abc* member is not yet defined.

2.4.3 Sources, Sink & Sanitizers Example

Sources are sources of data, such as data given as parameter to a *HTTP GET request*. These sources represent sensitive data, which, depending on the rules, we don't want to emit directly into the HTTP response. Sinks are destination points for the source data; when source data goes straight to a sink, without passing through a sanitizer, then there's a security flaw in the code. The sanitizer, as the name indicates, sanitizes the input, after which it is safe to be passed to a sink.

For example, a sink could be the HTTP response, which is emitted to the client. Given a GET parameter as source data, e.g., in *http://example.com/api?a=b a* is a GET parameter and *b* is its value, and the HTTP response as the sink, then the application will want to sanitize the source data before emitting it to the sink because otherwise a Cross-Site Scripting [4] attack might occur, i.e., a malicious attacker can inject arbitrary javascript into a benign website. In this example, a valid sanitizer would be one that encodes a string into a html-sanitized string, i.e., replacing '<' by '&lt;', etc.

[4] `http://en.wikipedia.org/wiki/Cross-site_scripting`

2.4.4 References

Although I had a clear idea of how I would have wanted to approach this project, I've used terminology as described in [1], [2], and [3], as these papers describe similar techniques but for different programming languages (*JavaScript*, *C*, and *Java*, respectively.)

2.5 Mentor from Mozilla

Finally, as this is a project originally mentored by Mozilla, there will also be a Mozilla guy involved, namely Stefan Arentz. Although we will have to come to an agreement between the different parties involved, this shouldn't give any additional problems, except for some extra communication for myself, but that's no problem.

Note that Mozilla will mainly be collecting sample data, e.g., misconfigured scripts, vulnerable scripts, etc. And will not, unless discussed and agreed upon by both parties, help with writing any code and/or part of the thesis (doh). (We will be using Git as revision control, so this isn't too hard to track.)

However, given the fact that this project will be modularized, other parties will be able to add new modules containing other, previously unsupported, web-frameworks, and/or other kind of security bugs which are not already included.

3 Security Hazards

This section discusses various security hazards which can be approached using the tool, how we can approach them, etc. Although unrelated to analyzing the source code itself, the tool will also support detection of e.g. misconfiguration, in order to make it an all-in-one tool. (Note that such scanners / modules can be generated by external people, due to the modularity of the project.)

3.1 Misconfiguration

An obvious but common mistake is one of the various types of misconfigurating a project. Configuration data can be stored in various locations; an external configuration file, an external python file, hardcoded inside a python file.

For external configuration files one can check the filetype, such as *ini* files, *python* files (in the form of *key = value* statements), or even *json* files. The information from these various kinds of configuration files can then be parsed as required.

3.1.1 Finding Configuration Files

First the module has to find all possible configuration files. It can find configuration files by recursively searching from the root directory through all subdirectories. Furthermore, python source files can be scanned for strings such as *"config.ini"*.

3.1.2 Preconfigured credentials

It is common for an application to have some sort of admin interface, in these cases there will be a password for the *root*, *admin*, or similar user. The password, however, may be anything but plaintext. For example, a project may have a *md5* hash as password, which is considered broken.

3.1.3 Hardcoded Keys

Some applications may use some sort of public/private key encryption, OAuth authentication, or other mechanisms to enforce trusted communication. Just like *preconfigurd credentials*, *hardcoded keys* should not be stored inside configuration files.

Specific submodules can be used to scan for certain thirdparty libraries (such as *OAuth*.)

3.1.4 False-positive configuration files

Libraries commonly ship with an example configuration file. In these cases, the source code should be scanned for statements which copy (or import) the example configuration file to the real configuration file (e.g., an application might use the example configuration file when it doesn't find a real one.)

Furthermore, it should be checked that an example configuration file doesn't contain real (or usable) entries. That is, an example configuration file might have the prototype to fill in real credentials for the library, but it should not contain weak and default credentials.

An incorrect example configuration file:

```
user = root
pass = toor
```

Whereas a correct example configuration file may look like the following:

```
# admin username
user =
# admin password
pass =
```

3.2 Pickle

Pickle is a Python library which allows one to store and load a Python object to or from memory, or file. For this particular reason it is known to be a vulnerable library among the Python libraries. Hence one should be very cautious to use it. This means that pickled data *must not* be loaded from untrusted data, such as that of a http request.

That being said, *Pickle* can be put directly on the list of Sinks.

3.3 Local File Inclusion

Local file inclusion occurs when an application loads a file from a path that is given through untrusted data, such as client data. Example attacks of this attack are reading configuration files; either specific to the library being used, or files such as */etc/passwd*.

3.4 Cross Site Scripting

Cross Site Scripting is when a website passes user-data unescaped back into a website. This results in a malicious user being able to craft a special URL which will execute some javascript, therefore being able to steal a users cookie [5], etc.

3.5 Cross Site Request Forging

Cross Site Request Forging is when an attacker can predict the URL query for a specific request/action upfront. In other words, when an application (or a specific part of it) is vulnerable to CSRF, then an attacker can embed such URL in for example a html image element, therefore giving instructions to the website without the user knowing about it.

An example request vulnerable to CSRF looks like the following:

```
# GET request to permanently remove a user from a website
/user/delete?user=jbremer&permanent=true
```

However, a website which has protection against CSRF requests, may have a page where the user can permanently delete the account using an URL like the one above, but it would include a unique token which is recreated every time the page is opened. For example, multiple requests to such page may result in the following URLs:

```
# first visit
/user/delete?user=jbremer&permanent=true&token=76847839753872398

# second visit
/user/delete?user=jbremer&permanent=true&token=79352769297489523
```

Due to the fact that the URL is randomized every time now, a malicious user cannot predict the URL, and will therefore not be able to trigger it using CSRF.

3.6 XSS & CSRF

Some websites might implement a correct mitigation against CSRF, but forget to protect against a combination of XSS and CSRF attacks. In this particular case, an attacker can use an XSS attack vector to do an *XMLHttpRequest* [6] (or similar) to the page on which the CSRF token can be found, and from there on do another request to the */user/delete* URL with the correct token.

A mitigation against this combination of attacks can be pulled off by checking the *Referer* of a request, and there's also some *Origin* HTTP headers, but both of these mitigations need some further research.

[5] http://en.wikipedia.org/wiki/HTTP_cookie

[6] http://www.w3.org/TR/XMLHttpRequest/

3.7 SQL Injection

SQL Injection vulnerabilities are often abused in order to steal various kinds of user data such as full names, email addresses, and even credit card information. SQL injections can be found by searching for strings. In a perfect world all the websites written in python use SQLAlchemy [7], which is not vulnerable to SQL injection. In other cases, the SQL queries will be checked. For example, many PHP applications are vulnerable to SQL injection because they modify the SQL query directly, rather than using so-called prepared statements [8]. An example of a vulnerable and a safe, prepared statement, follow:

```
# vulnerable SQL query
query = 'SELECT * FROM users WHERE id=' + user_data

# safe, prepared statement, SQL query
query = ('SELECT * FROM users WHERE id = ?', user_data)
```

The syntax of the prepared statement may differ between SQL libraries, but the question mark indicates an extra parameter, in this case *user_data*. This way the SQL query and the data are seperated, and therefore not vulnerable.

3.8 Captcha

When used, it may not be a good idea to use a homegrown captcha system, due to various reasons such as being too easy to defeat programmatically.

Furthermore, pages such as *register* pages, should usually be using a Captcha in order to prevent spammers from taking over the entire site (i.e., mass registering accounts, etc.)

3.9 Denial of Service

Certain sequences of code might be error prone to Denial of Service.

3.9.1 Expensive operations inside Web Handlers

When handling a request coming from the web, it is preferably fast. For example, actions such as reading (big) files, dns requests, and http request from inside a web handler function might be expensive and, when triggered a few thousand times, create a Denial of Service.

This counts especially for web handlers which operate without any kind of verification from the client (i.e., it's quite common for complex applications to have some handlers which perfrom quite some work when authenticated.)

3.9.2 Exceptions

Some web frameworks might not handle correctly when an exception is thrown in one of the page handlers, e.g., if a request to the URL */api/get* requires a GET parameter *id*, but this is not given, a dictionary lookup might throw an error. If a web-framework doesn't handle this correctly, e.g., it completely shuts down the website, then this is a serious Denial of Service bug.

[7]http://www.sqlalchemy.org/

[8]http://en.wikipedia.org/wiki/Prepared_statement

4 Internals

We've already outlined various techniques for analyzing the source code in the *Scientific Research* section.

4.1 SAT or SMT solver

Finally, we may want to proof that a sanitizer function a does in fact work as advertised, e.g., correctly html encodes an arbitrary string.

For this to work, we may be interested in using an SMT solver.

4.2 Modular

A very important feature of the project is to support various different frameworks and libraries. In order to keep all of this as modular as possible, each framework or library gets its own module (or maybe even multiple submodules.)

5 Conclusion

In this proposal I have introduced an interesting project, which has as far as I am aware not been done before (or atleast has not been made available to the public.) The aim of this project to research various ways to automatically and efficiently analyse websites implemented using Python web-frameworks for security vulnerabilities in order to improve the security of millions of people that rely on these website on a daily basis.

References

[1] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. *Saving the world wide web from vulnerable JavaScript. In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11).* ACM, New York, NY, USA, 177-187, 2011.

[2] Eric Bodden. *Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper. In Proceedings of the 7th Workshop on Programming Languages and Analysis for Security.* ACM, New York, NY, USA, 2012.

[3] Jürgen Graf, Martin Hecker, Martin Mohr. *Using JOANA for Information Flow Control in Java Programs — A Practical Guide* Programming Paradigms Group, 76131 Karlsruhe.

6 References

Secure Programming with Statis Analysis: `http://www.amazon.com/Secure-Programming-Static-Analysis-B` `dp/0321424778/ref=sr_1_1?ie=UTF8&qid=1357429171&sr=8-1&keywords=0321424778`

Google Results for Static Analysis Security: `http://scholar.google.de/scholar?hl=de&q=static+analysis+security&btnG=&lr`