BugBox : A Vulnerability Corpus for PHP Web Applications

Gary Nilson
Computer Science Department
University of Maryland

Kent Wills
Computer Science Department
University of Maryland

Jeff Stuckman
Computer Science Department
University of Maryland

Jim Purtilo
Computer Science Department
University of Maryland

Abstract

Web Applications provide a robust amount of code vulnerabilities that are exploited on a routine basis, but a large, diverse, automated, ease of use corpus for analyzing these vulnerabilities is not publicly available. BugBox, an open-source corpus and framework for PHP web application vulnerabilities was created to fill this void and in doing so encourage community support to enhance the quality of the project. We discuss the easy to use framework for web applications and further explain the motivation behind and application of BugBox. Through our framework implementation and corpus we can enhance testing vulnerability indicators and metrics, developing vulnerability definition representations, testing intrusion detection systems, or creating demonstrations for training purposes.

1 Introduction

Web applications are subject to a rich variety of exploit types, such as cross-site scripting (XSS), cross-site request forgery (XSRF), buffer overflow, and SQL injection. Many of these web applications are designed in PHP, for which Top Cyber Security Risks has reported to be a target for security threats [ref]. This variety and preponderance of exploits is logged extensively in popular exploit databases such as the National Vulnerability Database (NVD), Open Source Vulnerability Database (OSVDB), Common Vulnerability and Exposures Database (CVE), or the Exploit Database (EDB).

While many of these databases exist and disclose the details of the vulnerability and/or exploit, many do not provide a collection of structured vulnerable code that can be used for analysis. A suitable, structured collection of vulnerable code is typically not made publicly available and is scarce in quantity. The structured collection, often reffered to as a corpus, is useful in statistical analysis and hypothesis testing if it meets certain criteria. To elaborate on the suitability of a corpus, one most consider several of these criteria, which include: representativeness of the samples, quantity of the samples, diversity of the samples, and reproducability of the tests that produced the samples.

Uses of a corpus of this type include: serving as a training aid for security teams, penetration testing, testing static analysis tools, and metric based analysis. Projects such as "An empirical investigation into open source web applications' implementation vulnerabilities" by Huynh and Miller[1] could have benefitted from a corpus of this type. Insert our projects here.

Understanding the scarcity and labor involved in developing a corpus that is suitable, has led to the in-house creation of BugBox, a publicly available, scalable, efficient, easy-to-use framework and corpus that collects vulnerable code signatures through an automated trace based collection method. To increase the sample size and diversity of BugBox, we must attract community support. In order to successfully use this support, we provide a framework that is easy to maintain, quick to use, automated, fully compartmentalized, and scalable. Through the use of abstraction and encapsulation, the management of application, environment, and exploitation has been streamlined in BugBox. The design is intended to make it practical to manage a large database of exploits, along with their target environments. Our corpus is more than an exploit repository, it contains the software and its dependencies in which a vulnerability exists, the configuration of the software in it's vulnerable state, exploit code that will trigger the security breach, and data that describes any distinguishing attributes of the bug. [define attributes futher]

evaluating static analysis tools[2], penetration testing and training security teams with vulnerablity injection[3],

and computing attack surface metrics.[4]

2 Vulnerability Representation

A few effective methods for finding which piece of code creates a vulnerable condition in a program are line-based, run-based, and trace-based approaches. Henceforth, we will refer to these methods as *vulnerability localization*. Works by Fonseca and Vieira[5] use a line-based approach for vulnerability collection. This approach uses line numbers represented in program patches to identify the location of the vulnerabilities. Whil, in general, this is a sound approach, patches in a program may only mask the vulnerable code instead of replace it. Other by Song[6] create vulnerability signatures from intermediate byte code. This approach, while quick and efficient cannot be used for a PHP web application. The method proposed for our corpus is a trace-based collection approach in order to capture code vulnerabilities in software. The trace-based collection method is performed through monitoring the programs execution through the use of XDebug.

3 Requirements

BugBox is designed to work with the Debian GNU/Linux distribution and compatible distributions. It can be distributed as a self-contained virtual machine, or as a package that can be installed on an existing system. The machine must have sufficient storage (roughly 4 GB per OS environment, and 2 GB for the application, engine, and exploit sources [Confirm these #s]). Dependencies for BugBox include MySQL, Selenium Server, debootstrap, and the Advanced Packaging Tool (APT.)

4 Environment

In order to conduct tests in an isolated environment we use a virtual jail, the linux chroot environment. Staging the application in the chroot environment provides locality, reproducibility, and stability and flexibility.

The web application remains local. Many applications, such as wordpress, have a MYSQL backend. If we were to host the application on a different server or VM we would have to ensure that the MYSQL database is properly setup each time. With the chroot environment, the process is simplified, we keep a MYSQL database outside of the CHROOT Jail in order to facilitate MYSQL access.

Current tests are independent from future tests. We load a clean application into the chroot jail every time we wish to run a new test. This ensures that there is no corruption of the original web application and provides reproducible results when testing.

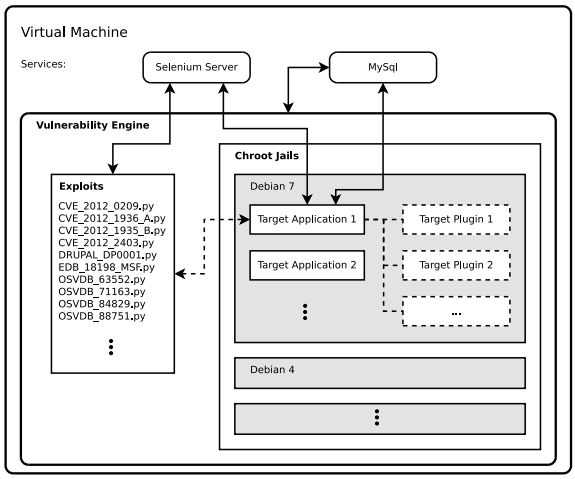

Figure 1: System Diagram

The web application cannot contaminate our testing environment. If a web application crashes due to the malicious script, we can ensure that it does not crash our corpus environment. In the worst case the chroot becomes corrupt for which we can treat in an isolated scenario tha does not have un-intended side effects in our testing environment.

The web application will never be contaminated. We store a separate backup in order to verify that the selenium scripts did not modify the Web Application in any way. If the Web Application is corrupt, then we simply copy the web application from the backup folder to the package folder. Future implementations will store only the md5 hash of the program on the corpus server and the backups on an external server. This will allow for a quick comparison followed by a remote copy for mismatched hashes. This separation will ensure that backups will never be corrupted through use of the corpus and expedite checking.

Applications with different OS environments can be run. If an application was built for debian 6 as opposed to debian 4, we can setup a chroot environment that has support for that specific OS. This gives us the flexibility to test the web application in whatever OS environment we choose.

5 Framework

At the core of the BugBox framework, the organization of the vulnerability engine is driven by the Python module and package system. The system breaks

down into the following four python modules: corpus.**Engine**, corpus.**Targets**, corpus.**Exploit**, and corpus.**SeleniumDriver**. Figure 1 illustrates the general structure of the corpus environment, with arrows showing lines of control or communication.

The **SeleniumDriver** is a wrapper class for the Selenium's Firefox web driver.

5.1 Vulnerability Engine

The **Engine** drives the environment setup, tear-down, and exploitation process. It contains most of the logic for doing work with exploits. deploying the web application in the chroot jail, starting the trace collection The corpus contains a main engine, written in python, to load a web application, gather traces, run exploits, and cleanup. We provide an engine to abstract away the details of the setup process so that contributors can focus on creating new exploit scripts. This separation accomplished two tasks, it makes developing for the corpus more approachable for programmers that want to contribute and provides a robust framework that can provide more of a challenge to seasoned programmers.

5.2 Target Module

The **Targets** module has as submodules each application and application plugin that are associated with an exploit. The anatomy of a "Target" module

5.3 Exploit

Exploit is the superclass for each exploit in the corpus, defining interfaces and attributes that the engine uses to manage the environment and exploitation.

Management actions, recording commands attributes,

```
__init__()
exploit()
check() [not yet]
cleanup()
```

5.3.1 As a Standalone Application

Each exploit is defined in it's own python file as a module. In Python, modules can either be imported from other modules, or executed directly from the command-line. The exploits written for BugBox take advantage of this property to give the researcher the option of writing scripts to do work on a set of exploits, or to invoke one exploit at a time. When run from the command-line, the supplied arguments are passed directly to the vulnerability engine, which supports the options shown below:

```
Usage: python OSVDB_89960.py [options]
Options:
    start:          Start exploit instance
    stop:           Stop exploit instance
    exploit:        Run the exploit
    check:          Check if the corresponding
                    environment is running
    xdebug_on:      Turn on xdebug autotrace
    xdebug_off:     Turn off and collect
                    xdebug autotrace
```

5.3.2 As a Python Module

Scripting of experiments. corpus.Query [Work in progress] interface for using querys of exploit metadata to manage sets of exploits/environments.

5.3.3 Selenium Scripting

We aim to gather code that is vulnerable. We drastically can reduce the amount of computation needed by reducing the code size that needs to be analyzed through the use of past, found exploits.

In order to better isolate vulnerable code, we create a selenium script in Python, replicating actions a user would take in performing an exploit on a web application. While the script is running, we can have XDebug monitor the execution and output the final execution trace for the selenium script execution. We can see future additions to this by turning XDebug on and off through cookie manipulation while the script is running, furthermore reducing the size of the vulnerable code.

By setting up the application loader engine, we are able to create concise python selenium scripts for data collection. The following code shows how easy it is to run an automated exploit:

Pros: -Ease of use, great python bindings -Uses browser's javascript engine -Good for Demonstration/visualization

Cons: -Not a necessary dependency, urrllib+cookielib would suffice -requires SeleniumServer to run as a service on the host system -Slower than crafting all requests directly -Many times, we still need to use auxiliary libraries to send specially crafted requests anyway

[figure of a session hijack in a web-browser?]

We are restricted to web applications because of the use of selenium scripts in our corpus. [GJN ADD NOTES FROM JEFF's PAPER ON THIS]

5.4 Trace Collection

Stuff about interaction with XDebug. XDebug is a feature-rich PHP debugger that can be used to easily collector traces enabling various run-time analyses.

```
import corpus

class Exploit (corpus.Exploit):

    def __init__(self):
        corpus.Exploit.__init__(self, {
                'Name' :        "CVE_2012_2403",
                'Description' : "Creates a post containing a XSS payload.",
                'References' :  [['CVE','2012-2403'],
                                 ['OSVDB','81463']],
                'Target' :      "Wordpress 3.3.1",
                'Type' :        "XSS",
                'VulWikiPage' : "http://seamster.cs.umd.edu/CVE-2012-2403"
                })
        return

    def exploit(self):

        payload = "<a href=\"#\" title=\"XSS http://example.com/onmouseover"
                  "=eval(unescape(/%61%6c%65%72%74%28%31%29%3b%61%6c%65%72"
                  "%74%28%32%29%3b%61%6c%65%72%74%28%33%29%3b/.source))//\"
                  ">XSS</a>"

        driver = self.create_selenium_driver()
        driver.get("http://localhost/wordpress/?p=1")
        driver.find_element_by_id("author").clear()
        driver.find_element_by_id("author").send_keys("selenium script")
        driver.find_element_by_id("email").clear()
        driver.find_element_by_id("email").send_keys("selenium@python.org")
        driver.find_element_by_id("url").clear()
        driver.find_element_by_id("url").send_keys("www.python.org")
        driver.find_element_by_id("comment").clear()
        driver.find_element_by_id("comment").send_keys(payload)
        driver.find_element_by_id("submit").click()
        driver.cleanup()

        return

if __name__ == "__main__": # Exploit invoked from command-line
    engine = corpus.Engine(Exploit())
    engine.parse_args(sys.argv)
```

6 Scalability

In order to create a system that can handle a growing amount of vulnerabilities that are independent from one another, we represent an exploit on an application through Selenium scripts, as noted in the Framework section. Selenium scripts allow us to perform operations on web applications without fear of working in an unintended, compromised environment.

7 Use Cases

By providing a corpus that explicitly logs the steps taken in accumulating the log files, we have more flexibility. This flexibility can be seen with the following example:

```
Jon is told by his advisor that he
needs to collect more trace data in
order to get a proper sample size for
his research.  John quickly creates
a selenium script for the exploit
```

```
he wants to collect and shows the
advisor his results.  The advisor was
generally happy with the trace data
that he collected, but instead wanted
him to do a slight modification to
the exploit.  If Jon did not have
the selenium script at hand, he would
have to duplicate all of the work
previously done.  Since he does have
the script on hand, he can quickly
make a change to the script and
re-run in seconds versus hours.
```

While the above process is only shown in one iteration, most students know that this is not the case. One hour of work can turn into a whole week of work without the proper framework in place. The above situation also shows that the selenium script can be discussed with the professor to show validity of the data and provide talking points for how the exploit was applied.

8 Acknowledgements

Metasploit, undergraduate summer labor, etc..
Now we're going to cite somebody. Watch for the cite tag. Here it comes . The tilde character (~) in the source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

9 Future Development

Distribution Virtual machine v.s. debian package. Prebuilt chroot jails v.s. build scripts. i.e. size ¡–¿ setup proceess balance. Striking a balance between

Services Why run Selenium/Mysql in VM vs a chroot jail?

Isolating attack event (with xdebug manually/cookies/etc...)

Selenium driver and aux modules Explore the possibility of a unified communication interface. This may be necessary in order to cleanly interact with xdebug with appropriate cookies set on a per-request basis (especially when modules other than Selenium are used for communication).

Payload standardization For each exploit currently in the corpus, there is no standard for the payload used in the attack. Since many studies may be sensitive to the payload type and encoding, it makes sense to provide the researcher with fine-grained control over this property. The Metasploit Framework has a very robust system for managing exploits along with their payloads and encodings, and can be a model for implementing this.

10 Conclusion

Everything is great, we just need community involvement to mature the framework and to help write more exploits.

11 Availability

[Available as a 10 GB virtual machine and as a standalone debian package?]

```
git://bugbox.github.com/blahblah

http://www.vulnerabilitywiki.com
```

References

Notes

[1]

[2] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code," ???

[3] J. Fonseca, M. Vieira and H. Maderia, "Traning Security Assurance Teams using Vulnerability Injection," 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing

[4] J. Stuckman and J. Purtilo, "Comparing and Applying Attack Surface Metrics," ???

[5] J. Fonseca and M. Vieira, "Mapping Software Faults with Web Security Vulnerabilities," International Conference on Dependable Systems & Networks: Anchorage, Alaska, June 24-27 2008

[6] Bitblaze