

BugBox : A Vulnerability Corpus for PHP Web Applications

Gary Nilson

*Computer Science Department
University of Maryland*

Jeff Stuckman

*Computer Science Department
University of Maryland*

Kent Wills

*Computer Science Department
University of Maryland*

Jim Purtilo

*Computer Science Department
University of Maryland*

Abstract

PHP Web Applications are a rich source of vulnerabilities due to their high exposure, diversity, and popularity. Accordingly, they are ideal specimens for empirical vulnerability research. There are not, however, publicly available resources of vulnerabilities that are both robust and practical to use. BugBox is an open-source corpus and framework for PHP web application vulnerabilities created to fill this void, and in doing so encourage greater quality security research. In this paper we describe Bug Box as a mechanism that encourages the distribution of vulnerability data and further supports research in developing vulnerability definition representations, testing intrusion detection systems, testing static analysis tools, and separately as a tool for training purposes.

1 Introduction

PHP Web applications are subject to a rich variety of exploit types, such as cross-site scripting (XSS), cross-site request forgery (CSRF), buffer overflow, and SQL injection. A recent study by White Hat Security [8] analyzed seven web application languages: ASP, ASPX, CFM (Cold Fusion Markup), DO (Struts), JSP, PHP, and PL (Perl) showing that PHP, while having the smallest attack surface in their tests, produced one of the highest average number of serious vulnerabilities per website. This variety of exploits in PHP and other languages 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 are easy to access and disclose the details of the vulnerability and/or exploit, they do not provide a collection of structured vulnerable code, a corpus, that can support statistical analysis

and hypothesis testing. The pursuit for such a corpus was driven through prior research, in developing attack surface metrics [7] and evaluating intrusion prevention systems [6], and is furthered through our continued research in dynamic decentralized intrusion detection. Furthermore, a corpus can facilitate a variety of activities of interest to the security community including evaluating static analysis tools [9], penetration testing and training security teams on vulnerability injection [2], and analyzing open source web applications [3].

BugBox is a framework that streamlines the collection and sharing of vulnerability data and facilitates cyber security experiments. This mechanism allows users to identify a vulnerability in a web application, quickly develop an exploit script, and collect execution traces of vulnerability data, all of which can be easily packaged and distributed to the community.

2 An overview of BugBox

At the core of BugBox is a packaging system that encapsulates vulnerabilities, vulnerable applications, and their exploits into self-contained, redistributable modules. These modules are automatically installed in an isolated `chroot` environment, which allows for the user to perform the exploit without incurring additional setup effort or the risk of contaminating a live system. These modules also provide metadata describing vulnerabilities and scripts which automatically execute exploits against the packaged applications. Because the vulnerabilities and exploits are automated through Python modules, large-scale security experiments can be set up by coding against the provided API. Collection of exploit execution traces is also automated, to support research in intrusion and vulnerability detection.

2.1 Application packaging and the chroot environment

A key responsibility of BugBox is to manage packaged vulnerable applications in an automated way. BugBox first prepares environments for the applications by setting up the operating system, web server, and other dependencies. Preconfigured applications can then be automatically installed in this environment, where their prerequisites are guaranteed to be present and they are in a state that is exploitable. These environments are constructed by setting up `chroot` jails on a Debian Linux host machine, providing a virtual environment with less overhead than a full virtual machine. A `chroot` jail supplies an isolated operating system installation (sharing the kernel of the host), allowing for applications, libraries, and configuration to be customized and discarded as needed. These characteristics are important for BugBox, as some applications have conflicting dependencies, and exploits should not be able to corrupt the host environment.

We create a `chroot` environment by running utilities supplied by Debian which generate a fresh environment and install packages from the official distribution repositories. Because this environment can be based on any Debian release, different versions of the application dependencies can coexist on a single host machine.

Applications that are installed in our environment are packaged in Python modules called *target* modules. Target modules include the vulnerable application's PHP files, the application's database, and metadata on the target's dependencies. The BugBox framework loads the target module into the `chroot` environment by identifying an environment that satisfies the module's dependencies, mounting the module into the environment's filesystem, and running the database scripts against the local MySQL installation. Modules are removed by removing the module's files from the environment. Because 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.

2.2 Exploit and vulnerability packaging

A second kind of BugBox module, the exploit module, contains information on a vulnerability and a scripted exploit. Exploit scripting is simplified, because applications in BugBox are installed in a predictable environment which is repeatedly reset to its original state. The format of a BugBox exploit resembles an exploit in the Metasploit framework, including the CVE number, the vulnerability type, and a link to the page on BugBox's wiki where the module is maintained. Below is an exam-

ple of this information:

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	WIKIHOST/CVE-2012-2403

These attributes serve to document the module's exploit and associated vulnerability. They are also used by the BugBox framework to perform processes across all available exploits, such as displaying information about exploits meeting certain criteria. These processes can be extended by scripts calling the BugBox API, which can grab collections of exploit instances based on regular expression matching against the attributes.

Scripted exploits are implemented as methods in exploit modules. These scripts leverage Selenium, a scriptable application that drives and automates web browsers such as Firefox. Selenium integrates nicely with the framework using stable and well-documented Python bindings, providing demonstration/visualization capabilities, JavaScript support, and easy interaction with HTML objects. The exploit code interacts with Selenium to perform pre-exploitation checks of the target and execute the exploit by automating a browser.

2.3 Collection of traces and vulnerability localization

A key question across many applications of empirical vulnerability research is how to represent which functionality, or piece of code accounts for the vulnerable condition in a program. In studying this problem, referred to as *vulnerability localization*, it is important to have a large quantity of structured data to test hypotheses. The desired properties of this structured data may vary by line of study, but in the BugBox framework the primary method proposed for capturing code vulnerabilities is through the collection of execution traces. We accomplish this task by collecting execution traces during exploitation using XDebug, and then manually highlight the program paths relevant to the vulnerability.

Line-based, run-based, and trace-based approaches are all effective types of vulnerability localization definitions. A line-based [1] approach for vulnerability collection uses line numbers represented in program patches to identify the location of the vulnerabilities. However, there are many ways to fix a vulnerability and a patch represents only one of these ways or may represent irrelevant changes. A run-based [5] approach creates vulnerability signatures from intermediate byte code. The drawback to this approach is that it does not isolate the lines of source code that represents the vulnerability. Ex-

ecution traces, on the other hand, where traces are classified as *potentially exploitative* or *non-exploitative* allow for great flexibility matching programs against vulnerability signatures. A drawback to this approach, is that to increase the accuracy of the data, currently one must manually refine the traces.

3 Advantages of chroot jail

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 that 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.

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 exploit scripts, as noted in the Framework section. Exploit scripts allow us to perform operations on web applications without fear of working in an un-intended, compromised environment. Scalability is critical because we ultimately want to do mass, automated experiments in tandem. **Introspection** Since the `chroot` environments exist on the host machine as simple directories, it is trivial to write scripts for managing the environment. This property allows for easy management of web applications, simple maintenance using incremental backups, and provides for other paths of research such as system taint analysis.

3.1 Developing BugBox modules

To add a new entry to the corpus one has to create two to four main entities. Writing an exploit, and generating

an execution trace are required, and depending on the target application, one may have to define a new target module. Sometimes, like when an application requires old versions of PHP or Apache, one may have to also setup a new target OS environment. However, because PHP web applications do not tend to have a large range of system dependencies, the majority of targets require at most a handful of different OS environments.

An exploit module in BugBox is simply a subclass of the framework's base exploit class, with various overridden methods and defined attributes that allow the Engine to do work with it.

The execution trace is generated using XDebug to capture the application's state throughout exploitation. XDebug is a feature-rich PHP extension that provides debugging and profiling capabilities. The BugBox engine instructs the debugger to capture execution traces by setting global environment variables in a configuration file prior to exploitation. Post-exploitation, the engine collects and organizes the traces on a per-session basis. XDebug also supports managing trace collection using cookies, which when implemented in BugBox, will allow further separating traces on a per-request basis.

To add a new vulnerability to the corpus, we first select an application vulnerability and determine whether the package is defined in the corpus, and proceed to create a target module otherwise. Although we focus primarily on open source applications, it is sometimes hard to obtain the vulnerable versions of an application. Therefore, creating the target module may involve hunting down patches in a mailing list or searching through the branches of a source repository. We then perform a manual execution of the exploit to verify that we understand the scope of the exploit, from which we proceed to write the exploit so that the process will be integrated into the corpus. Finally, we collect the trace of the program while our script is being executed and add it to our corpus.

4 An Example Execution of BugBox

The following Python scripts illustrate two use-cases for the BugBox framework. Both of these examples show ways to make use of vulnerability meta-data to run different experiments. In the first example, all XSS vulnerabilities existing in the the corpus are sequentially invoked, and execution traces are collected for analysis:

```

1 import config
2 from framework import Query, Engine
3
4 for Exploit in Query().get_by_type('XSS'):
5     engine = Engine(Exploit(), config)
6     engine.startup()
7     engine.xdebug_autotrace_on()
8     engine.exploit()
9     engine.xdebug_autotrace_off()
10    engine.shutdown()

```

Here, instead of collecting traces for all vulnerabilities of the same type, we are interested in running all exploits across all versions of a given application:

```

1 import config
2 from framework import Query, Engine
3
4 for Exploit
5     in Query().get_by_re('Target',
6                           'Wordpress.*"):
7     engine = Engine(Exploit(), config)
8     engine.startup()
9     engine.xdebug_autotrace_on()
10    engine.exploit()
11    engine.xdebug_autotrace_on()
12    engine.shutdown()

```

For simple jobs where the focus of study is a single vulnerability or vulnerable application, the run-time management of a single corpus entry can be done from the command-line. The most basic commands are provided in the *bbmanage.py* utility with the following options:

```

Usage: ./bbmanage.py [command] <options>
Commands:  Options:
list       <exploits | targets |
           types | running>
info       <exploit_name>
start      <exploit_name>
exploit    <exploit_name <display>>
stop       <exploit_name>
trace_on   <exploit_name>
trace_off  <exploit_name>
autorun    <exploit_name>

```

The process of setting up an OS environment, configuring the target application, interacting with the PHP debugger, and all post-job cleanup occur behind the scenes in the BugBox framework. Any of these tasks would have taken hours to accomplish using our previous approach, which was based on manually managing various virtual machine snapshots. Using Python as a layer of abstraction above the OS, target application, and exploits, it is now possible to easily automate research tasks. The following sections give an overview of the system design that makes this abstraction possible.

It is the job of the **Engine** to then match exploits with target modules and actually perform setup, configuration, and teardown work. The Engine will actually mount the web application's folders within an associated chroot jail, load a stored MySQL database, install the appropriate

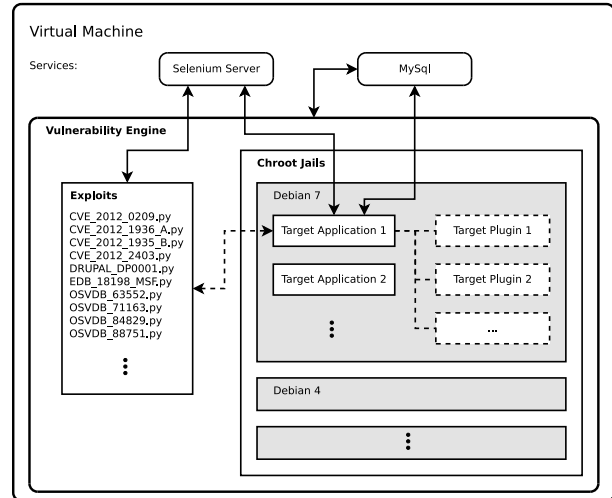


Figure 1: System Diagram

XDebug PHP debugger configuration file, start the web server, and eventually perform cleanup operations associated with these steps.

5 Framework

Figure 1 illustrates the structure of the corpus, with boxes and arrows showing lines of control or communication. The framework breaks down logically into the following five components: **Host Environment** (currently the Virtual Machine), **Engine**, **Target Environment**, **Target**, and **Exploit**. The diagram shows that the corpus is composed of any number of exploits, chroot jails, target applications, and associated target plugins, that exploits and chroot jails are under control of the vulnerability engine, the exploits interact directly with the target application (during exploitation) both through the Selenium Server and directly using libraries such as *urllib*, all databases associated with the target applications are hosted directly on the host environment, and the database server is under control of the vulnerability engine.

Except for the host environment, each of the five major components are represented directly in BugBox as either a Python module or a package. The design is intended to make it practical to manage a large database of exploits, along with their target environments. For each exploit, all relevant details of a vulnerability are maintained for the researcher. A given corpus entry we include the software in which a vulnerability exists, the configuration of the software in its vulnerable state, exploit code that will trigger the security breach, and any relevant meta-data.

5.1 Engine

The **Engine** drives the environment setup, tear-down, and exploitation process. The engine was created to abstract away details so that researchers can easily run exploit scripts without having to worry about getting the target application running every time.

The vulnerability engine controls the actions of a given exploit, and ensures that the proper target environment, application, and application plugins are properly setup in the `chroot` environment. The role of the engine is illustrated in the two use-cases. For example, following relevant lines in Use-Case 1:

Line 5 `engine = Engine(Exploit(), config)`

A new engine instance is constructed with an exploit instance and system configuration as arguments.

Line 6 `engine.startup()`

The engine creates a `chroot` environment on the host system to which the target application is copied and the pertinent MySQL databases are established. The setup of the environment is determined by the 'Target' attribute in the exploit instance.

Lines 7 & 9 `engine.xdebug_autotrace.on()`
`engine.xdebug_autotrace.on()`

The engine can modify the state of the trace collection in X_Debug to ensure that we collect only the trace pertinent to the exploitation.

Line 10 `engine.shutdown()`

Cleanup unmounts the `chroot` environment and returns the corpus environment back to an unaltered state.

5.2 Targets

The **Target** module controls all the application targets that the exploits may be applied to. More specifically, this represents web applications such as WordPress, SimplePHPAgenda, etc. A copy of the target program with a typical configuration, is always loaded before any exploit script is run. Init scripts are written per application to provide the **Engine** with the proper details for the setup, for example:

```
1 | name = "Wordpress 3.3.1"
2 | application_dir_mapping = ...
3 |     [get_path("application"), "/var/www"]
4 | database_filename = get_path("database.sql")
5 | database_name = "wordpress_3_3_1_A"
6 | chroot_environment = "Debian7"
```

Some large web applications, such as WordPress, have many plugins that in themselves, have vulnerabilities. Therefore **Target** modules also allow for the inclusion

of plugins in the exploit framework. Both targets, and target plugins are resolved against the exploit attributes by the Engine during the setup process.

5.3 Exploit

The **Exploit** class is the superclass for each exploit in the corpus, defining interfaces and attributes that the engine uses to manage the environment and exploitation. The structure of Exploit is designed so that exploits can be written as a concise code statement. Each exploit is simply a subclass of Exploit residing in its own Python file in which all actions taken by the exploit are clearly visible. In addition, a Query module is provided that will return a set of Exploit instances based on matching against the meta-data specified in the exploit (as shown in the use-cases).

This separation accomplishes a step-like approach for getting involved with the corpus, where developing exploits for the corpus is more approachable and quick for novice programmers that wants to contribute and a robust framework that can be understood incrementally for larger contributions to the framework.

This template ensures that the selenium driver is properly initialized bound to the exploit and makes sure that an exploit script has been defined as well as defines any specific actions on tear-down.

The exploit module is annotated by its attribute dictionary. This structure is inspired by the exploit class from the Metasploit [4] module. In it we declare the name, description, references, target, type and wiki page for the exploit by the attributes dictionary. The name is the technical name specified in one of the online databases of exploits and vulnerabilities mentioned earlier in the paper. The description is a brief statement of what the exploit is supposed to do, because the name is typically not descriptive enough for a person to understand what type of exploit is being applied. The Target is the web application target that is being used for the exploit. The type is the type of exploit being conducted. The VulWikiPage is the wiki page setup for an author to place auxiliary information about the exploit or script.

Here is an example body of an exploit procedure used for the simplest of XSS attacks. The entire exploit consists only of instantiating the selenium driver and submitting a post on the WordPress site with the payload:

```

1 | payload = "<a href=\"#\" title=\"XSS http:\" \
2 |           "//example.com/onmouseover=eval(\" \
3 |           \"unescape(/%61%6c%65%72%74%28%31\" \
4 |           \"%29%3b%61%6c%65%72%74%28%32%29%\" \
5 |           \"3b%61%6c%65%72%74%28%33%29%3b/.\" \
6 |           \"source))/\">XSS</a>\"
7 |
8 | driver = self.create_selenium_driver()
9 |
10 | driver.get("http://localhost/wordpress/?p=1")
11 | get = driver.find_element_by_id
12 | get("author").send_keys("selenium script")
13 | get("email").send_keys("selenium@python.org")
14 | get("url").send_keys("www.python.org")
15 | get("comment").send_keys(payload)
16 | get("submit").click()

```

5.4 Cleanup and maintenance

6 Lessons learned

This wrapper class allows the exploit writer to choose whether to display the Firefox browser during exploitation. This feature has proven very useful for debugging exploits, and has lead to quicker turn-around in the exploit development. With the Selenium Driver, one can also orchestrate demonstrations to teach others the principles behind web vulnerabilities. A setup could entail showing a live feed of tcpdump and the Apache logs on one screen, while in real-time one Firefox process shows the actions taken by an attacker posting a malicious script on a forum, after which the session for an unsuspecting admin in an adjacent Firefox process shows the activation of a CSRF exploit.

Some research projects conducted in our software engineering group at the University of Maryland are centered around collection of metrics data for vulnerabilities. Since we were unable to find a corpus that suited our needs, we started to build an in-house corpus. Originally our corpus architecture was a VM based.

The idea was to be able to mount web application targets onto VMs. We had a similar idea in th sense that the VM's could be setup and brought down, saving the current state that they were in. Once the application was mounted, we performed an exploit manually and saved a snapshot of the VM instance. All exploits applied to "tainted" applications, so we had to be careful that one exploit did not interfere with another. We hired summer students to create exploits and save new snapshots, from which we had them report their progress on the wiki we had built.

We found that because the process was not sufficiently documented, it was harder to verify that the exploit was executed in the correct manner. Many times pages that were logged as exploited, did not show the representation of the exploit. Furthermore it was not possible to perform automated query-able tests for the collection of exploits.

7 Future work

Future development on the framework will include features for controlling trace collection through cookie manipulation via XDebug. This will reduce the collection of insignificant interactions and provide a more refined break-down of the exploitation process since traces can be grouped by HTML request. In order to achieve this, we may have to explore the possibility of forcing the exploit writer to use a single communication interface, instead of the Selenium/cookielib/urllib combination in the current approach. 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).

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.

BugBox is designed to work on the Debian GNU/Linux and compatible distributions. It will be distributed both as a self-contained virtual machine, and as a package that can be installed on an existing system. The machine must have sufficient storage (roughly 4 GB per OS environment, and up to 2 GB for the application, engine, and exploit sources), with access rights to use run Linux chroot jails. Dependencies for the BugBox host system include MySQL, Selenium Server, debootstrap, the Advanced Packaging Tool (APT.), and any minimum hardware requirements for running the target web applications.

Community contributions will be key to building a corpus of sufficient size, and the BugBox framework code will be made generally available under an open-source license. Because the corpus is composed entirely of open-source applications, target modules are distributed under the application's respective license. In addition, we maintain a vulnerability wiki detailing the entries in the corpus, and a web-interface for the git repository that where the code is managed. Collaboration will be coordinated through the issue tracking system on the website, with git, and through the mailing list.

Acknowledgments

The authors thank the United States Office of Naval Research for its support for this research.

References

- [1] FONSECA, J., AND VIEIRA, M. Mapping software faults with web security vulnerabilities. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (2008), pp. 257–266.
- [2] FONSECA, J., VIEIRA, M., MADEIRA, H., AND HENRIQUE, M. Training security assurance teams using vulnerability injection. In *Dependable Computing, 2008. PRDC '08. 14th IEEE Pacific Rim International Symposium on* (2008), pp. 297–304.
- [3] HUYNH, T., AND MILLER, J. An empirical investigation into open source web applications' implementation vulnerabilities. *Empirical Software Engineering* 15, 5 (2010), 556–576.
- [4] RAPID7. Metasploit framework retrieved from <http://www.metasploit.com/>, 2013.
- [5] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 1–25.
- [6] STUCKMAN, J., AND PURTILO, J. A testbed for the evaluation of web intrusion prevention systems. In *Proceedings of the 2011 Third International Workshop on Security Measurements and Metrics* (Washington, DC, USA, 2011), METRISEC '11, IEEE Computer Society, pp. 66–75.
- [7] STUCKMAN, J., AND PURTILO, J. Comparing and applying attack surface metrics. In *Proceedings of the 4th international workshop on Security measurements and metrics* (New York, NY, USA, 2012), MetriSec '12, ACM, pp. 3–6.
- [8] WHITEHAT. Website security statistic report 9th edition retrieved from https://www.whitehatsec.com/assets/wpstats_spring10_9th.pdf, 2010.
- [9] ZITSER, M., LIPPMANN, R., AND LEEK, T. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 97–106.