BugBox : A Vulnerability Corpus for PHP Web Applications

First Author *Institution*
Second Author *Institution*
Third Author *Institution*
Fourth Author *Institution*

Abstract

Web applications are a rich source of vulnerabilities due to their high exposure, diversity, and popularity. Accordingly, they are attractive subjects for empirical vulnerability research. However, there are no structured, public available vulnerability datasets that tie vulnerabilities to their applications and exploits. BugBox is an open-source corpus and exploit simulation environment for PHP web application vulnerabilities created to fill this void and facilitate quality security research. The packaging mechanism provided by BugBox encourages the distribution and sharing of vulnerability data, supporting research in vulnerability prediction and the testing of intrusion detection systems and static analysis tools. BugBox also facilitates developer education by demonstrating exploits and providing a sandbox where they can safely be attempted.

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 [5] 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 need for such a corpus was noted through our prior research in security metrics and the testing and evaluation of security products, 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 [6], 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, facilitates cyber security experiments, and furthers education on security vulnerabilities. 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. Exploits can also be demonstrated by visualizing a live web browser that shows the exploit being performed from the attacker's perspective. Advanced users and researchers can perform large-scale vulnerability experiments in an environment that can easily be guarded from contamination.

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. BugBox remains scalable as new modules for applications and exploits are developed, by deploying just one application at a time in the target environment

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 automated 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 the `debbootstrap` utility supplied by Debian which generates a fresh installation 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, there is no corruption of the original web application, providing 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 example 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

An important activity when performing empirical vulnerability research is representing which functionality, or piece of code, accounts for the vulnerable aspect of the program. We refer to this task as *vulnerability localization*, and BugBox is designed to collect structured trace data that assists in the localization process. The BugBox environment includes an execution trace collection facility supported by `XDebug`, which automatically gathers traces of the PHP application's execution while the exploit takes place.

We categorize the vulnerability representations used in vulnerability research as being *line-based*, *run-based*, or *trace-based*. A line-based approach for vulnerability representation associates particular lines of a program with a vulnerability. This data can easily be collected by mining vulnerability data from program patches [1]; however, there can be multiple ways to fix a vulnerability, and a patch represents only one of these ways or may contain irrelevant changes. Run-based representations and signatures [4] characterize vulnerabilities in terms of the inputs that may exploit them, rather than the lines of code that were faulty.

We are developing a third, *trace-based* approach for representing the locations of vulnerabilities. Execution traces of individual requests are classified by a regular language as *potentially exploitative* or *non-exploitative*. This approach allows for representations to be crafted that include the key source code artifacts that are invoked during exploitative runs, capturing each opportunity where sanitization or other defensive programming could have prevented the attack. By collecting execution traces of exploits, BugBox facilitates the construction of run-based or trace-based representations.

3 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. Although the focus is 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. Next a manual execution of the exploit is performed to verify the vulnerability report's claims and that the exploit mechanism is understood. Subsequently, the exploit module is written so that the entire process is recorded as an entry in the corpus. As a final step, the exploit is run against the target while the debugger collects the trace of the program after which it is added the corpus.

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.

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 metadata 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:

```
import config
from framework import Query, Engine

for Exploit in Query().get_by_type('XSS'):
   engine = Engine(Exploit(), config)
   engine.startup()
   engine.xdebug_autotrace_on()
   engine.exploit()
   engine.xdebug_autotrace_off()
   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:

```
import config
from framework import Query, Engine

for Exploit
    in Query().get_by_re('Target',
                         'Wordpress.*"):
    engine = Engine(Exploit(), config)
    engine.startup()
    engine.xdebug_autotrace_on()
    engine.exploit()
    engine.xdebug_autotrace_on()
    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

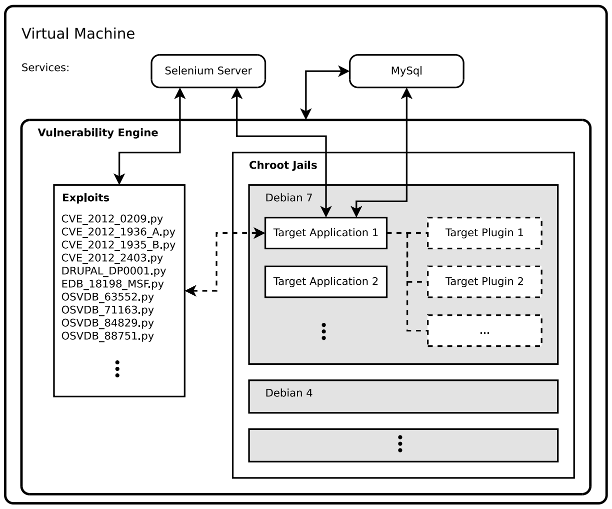

Figure 1: System Diagram

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 BugBox system, 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 associates 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 it's 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:

```
name = "Wordpress 3.3.1"
application_dir_mapping = ...
     [get_path("application"), "/var/www"]
database_filename = get_path("database.sql")
database_name = "wordpress_3_3_1_A"
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 it's 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 an easy approach to get involved with the corpus, making 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. In it we declare the name, description, references, target, type and wiki page for the exploit by the attribute 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 wiki page directs an author to record 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:

```
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")
get = driver.find_element_by_id
get("author").send_keys("selenium script")
get("email").send_keys("selenium@python.org")
get("url").send_keys("www.python.org")
get("comment").send_keys(payload)
get("submit").click()
```

5.4 Cleanup and maintenance

Multiple methods that perform cleanup and maintenance are included in BugBox to ensure that results are reproducible. With this in mind, there are many changes that occur in the system through the course the exploit cycle that need to be eliminated. Even changes that would normally seem innocuous, like recording events to a log-file, have the potential to at some point change program behaviour, and should not be allowed. BugBox has separate methods that maintain the integrity of the target modules the environments.

Since the `chroot` environments exist on the host machine as simple directories, it is trivial to write scripts for managing them. Scripts using `rsync` are employed to periodically overwrite the `chroot` jails to maintain the integrity of the operating system. Each exploit is then responsible for managing it's own side-effects on the target application. For instance, many exploits result in the inclusion of an arbitrary file to be executed by the webserver. The exploit superclass provides for a cleanup method to be written when implementing an exploit for exactly these types of vulnerabilities. When defined, the cleanup method is invoked after the successful application of exploit to restore the target module to it's pre-exploitation state.

This wrapper class allows the exploit writer to choose whether to display the Firefox browser during exploitation. This feature has been 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.

6 Lessons learned

In developing BugBox, we learned *four* key lessons: Flexibility of environment, the use of visual aids in speeding up exploit script development, developing a large diverse corpus is quicker with community involvement, and the use of quick script verification to increase corpus quality and repeatability of experiments.

Having flexibility over choosing the environment in which web applications are installed is a critical factor in corpus development. The goal of creating a corpus is to create a wide variety of examples, and in doing this, we sometimes have to use web applications that are outdated. Incorporating these web applications into the corpus leads to dependencies on older language versions that have dependencies for different Linux distributions.

Exploit script development is a repetitive and necessary process. We found that the instant feedback provided by the Selenium visual framework allowed us to more rapidly create exploit scripts. This increase in productivity is due to the visual execution monitoring of the scripts, allowing developers to efficiently pinpoint errors, enhancing script generation.

In collecting samples for our original corpus, we failed to focus on collecting a representative sample across independent variables such as vulerability type and age. Because we intend for our corpus to be representative and suitable for statistical analysis, we are remediating this by collecting new vulnerabilities and stratifying the sample across these variables. We recommend that future teams collecting vulnerability data introduce proper sampling procedures as early in the process as possible.

7 Future work

We plan to implement additional 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. For each exploit currently in the corpus, there is no standard for the payload used in the attack. Since studies on intrusion detection or prevention systems may be sensitive to the payload type and encoding, researchers may desire finer-grained control over this property. The Metasploit Framework has a robust system for managing exploits along with their payloads and encodings and serves as a model for implementing this.

We also plan to make BugBox available to the security community in the near future. BugBox is designed to work on the Debian GNU/Linux and compatible distributions. The framework will be distributed under an open-source license, both as 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`. Community contributions will be key to building a corpus of sufficient size, and we also intend to deploy a wiki allowing contributers to download, submit and discuss BugBox exploit and target modules. Because the corpus is composed entirely of open-source applications, target modules will be distributed under the application's respective license.

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] 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.

[5] WHITEHAT. Website security statistic report 9th edition retrieved from https://www.whitehatsec.com/assets/wpstats_spring10_9th.pdf, 2010.

[6] 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.