BugBox : A Vulnerability Corpus for PHP Web Applications

Gary Nilson
Computer Science Department
University of Maryland

Kent Wills
Computer Science Department
University of Maryland

Jeffrey Stuckman
Computer Science Department
University of Maryland

James Purtilo
Computer Science Department
University of Maryland

Abstract

Web applications are a rich source of vulnerabilities due to their high exposure, diversity, and popularity. Accordingly, web application vulnerabilities are useful subjects for empirical security research. Although some information on vulnerabilities is publicly available, there are no publicly available datasets that couple vulnerabilities with their source code, metadata, and exploits through an executable test environment. We describe BugBox, a corpus and exploit simulation environment for PHP web application vulnerabilities. BugBox provides a test environment and a packaging mechanism that allows for the distribution and sharing of vulnerability data. The goal is to facilitate empirical vulnerability studies, security tool evaluation, and security metrics research. In addition, the framework promotes developer education by demonstrating exploits and providing a sandbox where they can be run safely. BugBox and its modules are open-source and available online, and new modules may be contributed by community members.

1 Introduction

PHP web applications are subject to a wide 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 [20] analyzed seven web application platforms: ASP, ASPX, ColdFusion, Struts, JSP, PHP, and Perl, concluding that, on average, PHP applications had one of the highest numbers of serious vulnerabilities relative to the application's attack surface.

A variety of approaches can be taken to mitigate vulnerabilities. Simple developer education can address many issues before they occur. Also, various software tools and approaches exist like vulnerability scanners, specially hardened platforms or libraries, and software metrics for risk estimation. All of these approaches can be enhanced with data gleaned from past vulnerabilities, bringing a greater empirical assurance that the technique works as intended. For example, developers can gain insights on security by experimenting on vulnerabilities that occurred in the past, while tool developers use historical defect data to evaluate their tools and motivate future improvements [21]. In our own research in security metric evaluation [18], we observed that existing sources of vulnerability data did not supply the necessary structure or metadata to evaluate them completely. This motivated us to build a corpus of *executable* vulnerabilities that is linked to an automated exploit and test mechanism, allowing for exploits to be observed and instrumented at any point during the data collection and experimentation process.

BugBox is a framework that streamlines the collection and sharing of vulnerability data, facilitates cyber security experiments, and furthers education on security vulnerabilities. It allows users to identify a vulnerability in a web application, quickly develop an exploit script, and collect execution traces of vulnerability data. The results are packaged into a structured format that facilitates sharing and collaboration. Exploits can also be demonstrated by displaying a live web browser visualization that shows the exploit being performed from the attacker's perspective.

Security researchers and tool developers can use BugBox to evaluate vulnerability mitigation techniques and perform large-scale experiments in an environment that can easily be guarded from contamination. The framework also facilitates the generation of vulnerability runtime data and measurements tailored to specific uses, supporting efforts to characterize malicious traffic and validate security metrics. Developers and system administrators could also use BugBox to learn about vulnerabilities or aid in evaluation of intrusion detection or prevention techniques. Finally, the BugBox framework and its vulnerability modules are freely available, facilitating future, collaborative development of additional modules or the framework itself.

2 Related work

The vulnerability modules for BugBox leverage existing databases of exploits and vulnerabilities, such as the National Vulnerability Database [10], the Open Source Vulnerability Database [12], and the Exploit Database [11]. While these databases are easy to access and contain significant details of the vulnerability, they do not provide a structured collection of vulnerable code that can support statistical analysis and hypothesis testing. By supplying this corpus and an associated test environment, BugBox compliments these databases by aggregating exploit procedures and data on a subset of the vulnerabilities they contain. Another complimentary project, the SAMATE reference dataset [9], provides vulnerability data for the evaluation of vulnerability discovery methods. BugBox covers a smaller number of vulnerabilities than SAMATE in greater detail, coupling each vulnerability to an automated exploit environment and guaranteeing that the full source code for each application is present.

Other efforts such as SecuriBench [8] and WebGoat [13] collect vulnerable software for research or educational purposes, although the vulnerabilities are not in the form of a structured corpus. Metasploit [14], a framework for building and packaging exploits, is used internally by some BugBox modules to exploit their associated vulnerability, although Metasploit modules themselves do not contain vulnerable applications or other vulnerability metadata. Various other non-security bug prediction research efforts [4, 22] also aggregate various metadata or source code from past defects.

A corpus of vulnerabilities coupled to a test mechanism, such as BugBox, could benefit various kinds of security research. Static and dynamic analysis tools for discovering vulnerabilities [2, 1] can be evaluated against previously discovered vulnerabilities [7], which BugBox provides for web applications. Attack prevention techniques relying on runtime detection of exploits, such as [19] and our own ongoing research in dynamic decentralized intrusion detection, would be especially suitable for evaluation with BugBox, as the test mechanism and automated exploit scripts are leveraged in these cases.

Finally, the corpus of vulnerabilities provided by BugBox can facilitate observational studies of vulnerabilities [6] and empirical evaluations of metrics for vulnerability prediction [15, 18]. The process of computing metrics against a corpus was explored in more detail in our previous wiki corpus research [16]. In these cases, having the entire source code of vulnerable applications is essential, as the application's vulnerable code must be compared with the application's entire code base.

3 An overview of BugBox

At the core of BugBox is a packaging system that wraps vulnerabilities, vulnerable applications, and their exploits into redistributable modules. These modules are automatically installed in an isolated `chroot` environment, that allows for the user to execute the exploit without manually installing the application or risking contaminating a live system. Because only one application is deployed in the target environment at any given time, BugBox remains scalable as new modules are developed and installed.

Modules also contain metadata that encodes properties of the vulnerability, including an identifier associating each exploit with its entry in one of the public vulnerability databases. The included exploits are in the form of scripts that automatically execute the exploit against the application installed in the local target environment. Because vulnerability installation and exploitation is automated through Python modules, large-scale automated security experiments can be run by calling the relevant functions in the provided API. The collection of exploit execution traces is also automated, easing the process of localizing vulnerabilities and exploits to particular sections of source code.

The exploit modules can also display the web browser during exploitation, depicting the actions taken by the Selenium driver through Firefox. This feature has been useful for debugging exploits, and has led to quicker turn-around in exploit development. This also allows for demonstrations that teach others the principles behind web vulnerabilities. Such a setup could entail showing a live feed of tcpdump and Apache logs on one screen, while one Firefox window shows the actions taken by an attacker posting a malicious script on a forum, and another Firefox window shows how an unsuspecting administrator can activate a CSRF exploit and execute the script.

3.1 Application packaging and the `chroot` environment

A key function of BugBox is to manage packaged vulnerable applications in an automated way. BugBox provides an environment for the applications by creating an operating system, a web server, and other dependencies. Preconfigured applications can then be automatically copied in this environment, so 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 VM (virtual machine).

A `chroot` jail supplies an isolated operating system installation (sharing the kernel of the host), allowing for applications, libraries, and configurations to be customized and discarded as needed. These characteristics are important for BugBox, as some applications have conflicting dependencies, and exploits could 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. These dependencies include software such as Apache, PHP, and MYSQL, and their dependant system libraries from a variety of vintages, as some older versions of applications depend on older releases of this dependant software.

The chroot jail meets BugBox's needs for isolation and state management in a more lightweight manner than creating a full VM. The operating system virtualization features provided by VMs are not needed, as PHP web applications (and most of their vulnerabilities) are typically platform independent. While VMs have superior capabilities for creating system snapshots and performing state recovery, they consume considerable RAM resources, and starting, stopping, and managing them is more cumbersome. Backing up and periodically cleaning up the environment with `rsync` was found to be adequate for restoring the system to its unadulterated state, eliminating the need for VM snapshots.

`Debootstrap` and related tools allow the BugBox environments to be distributed as a set of scripts, allowing users to build the whole environment from a single set of scripts. The scripts automatically install the necessary Debian packages from an official mirror, and they perform the necessary configuration so the environment is ready to run the web applications. Each environment is a single folder which contains a complete Debian installation. The environment can be maintained by using basic file IO operations, allowing scripts to install and remove vulnerabilities by copying and deleting files and avoiding the delays that would be associated with rebooting and rolling back a VM. Although multiple vulnerabilities cannot be concurrently installed in a single BugBox environment, removing one application and installing another can be accomplished quickly and automatically.

Because BugBox hosts applications containing known vulnerabilities, it is recommended for it to be deployed in an environment isolated from the internet. This ensures the security of the system and that any measurements taken are free from the influence of external traffic. If such an environment is not readily available, BugBox can be installed on a single VM, which can have its internet-facing network interfaces removed after installation. As several BugBox chroot jails can coexist on one VM, an existing VM can be used for this purpose if one is available.

3.2 Exploit and vulnerability packaging

Applications that are installed in the BugBox 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 a `chroot` environment by identifying a Debian version 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 a new test is run, there is no corruption of the original web application, providing reproducible results when testing.

A second kind of BugBox module, the exploit module, contains information on a vulnerability and a scripted exploit. Developing BugBox-specific exploit scripts is considerably simpler than writing general-purpose exploits, because applications in BugBox are installed in a predictable environment which is repeatedly reset to its original state. The format of a BugBox exploit module resembles an exploit in the Metasploit framework, including the CVE number, the vulnerability type, a link to the page on BugBox's wiki where the module is maintained, and other metadata. Below is an example of this metadata:

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 batch 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, grabbing collections of exploit instances based on regular expressions matching 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 with the framework through 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 executes the exploit by automating a browser.

3.3 Collection of traces and vulnerability localization

An important activity associated with empirical vulnerability research is representing which functionality, or section 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 collector supported by `XDebug`, which automatically traces the PHP application's execution while the exploit takes place.

We categorize the vulnerability representations previously used in 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 [5]; however, there can be multiple ways to fix a vulnerability, and a patch represents only one of those ways and may contain irrelevant changes. Run-based representations and signatures [3] characterize vulnerabilities by describing the inputs that may exploit them, rather than the lines of code that were faulty.

We are developing a third, *trace-based* approach for localizing 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 source code artifacts that were 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.

4 Adding vulnerabilities to BugBox

To add a new vulnerability to the corpus, several artifacts must be compiled. A new exploit script and metadata are always required. If a module does not yet exist for the target application, a new target module must also be developed. Although the BugBox corpus compiles vulnerabilities from open-source applications, it is sometimes hard to obtain the vulnerable versions of an application. Therefore, creating the target module may involve hunting down application versions from third-party website or searching through the branches of a source repository. Once the application has been obtained, a manual execution of the exploit is performed to verify the vulnerability report's claims and that the exploit mechanism is fully understood. Subsequently, the exploit script is written so that the entire process can be recorded in an entry in the corpus. As a final step, the exploit is run against the target while the debugger collects the execution trace of the program, which is then added the corpus.

Some vulnerable applications require old versions of dependencies (such as PHP or Apache) not present in the target environment, necessitating a new operating system instance. Because, as we have verified, some older Linux distributions can run under a newer kernel, this can be accomplished by simply creating a new `chroot` environment. Because most PHP web applications do not have a wide range of system dependencies, the application targets will only require a handful of different OS environments.

5 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 perform different tasks. In the first example, all XSS vulnerabilities in the the corpus are sequentially exploited, 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 (for example, to test an intrusion detection system which was installed in the test environment before running this process):

```
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 tasks where the focus of study is a single vulnerability or vulnerable application, the application can be managed from the command line. Some basic commands are provided in the *bbmanage.py* utility with the following options:

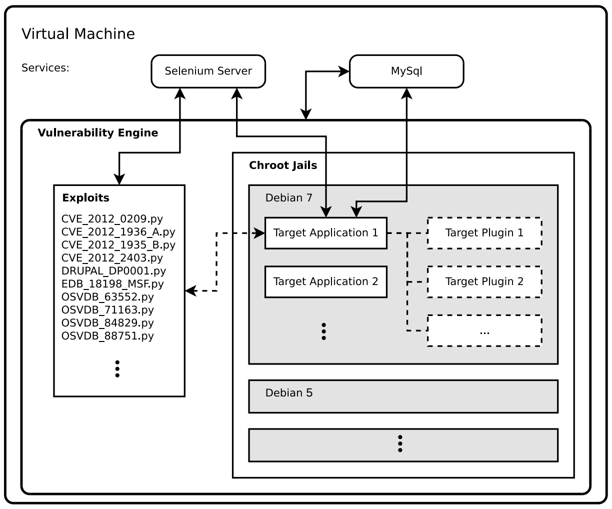

Figure 1: System Diagram

```
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>
```

6 Framework

Figure 6 illustrates the structure of the BugBox system, with arrows representing lines of control or communication. The framework contains five components: **Host Environment** (the physical or virtual machine running BugBox), **Engine**, **Target Environment** (the `chroot` environment), **Target Module**, and **Exploit Module**. The corpus is composed of exploits, `chroot` jails, target applications, and associated plugins for target applications. Exploits and `chroot` jails are under control of the engine, and the exploits interact directly with the target application (during exploitation) through the Selenium Server or 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 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.

6.1 Engine

It is the job of the **Engine** to match exploits with target modules and actually perform setup, configuration, and teardown work. The Engine will prepare a `chroot` environment by mounting target operating system and web application folders under the environment, load a stored MySQL database, install the appropriate XDebug PHP debugger configuration file, start the web server, and eventually perform cleanup operations associated with these steps.

To illustrate the role of the engine, the following lines in the first use case are relevant:

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. The choice of application 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 only the trace pertinent to the exploitation is collected.

Line 10 `engine.shutdown()`
This cleanup code unmounts the `chroot` environment and returns the corpus environment back to an un-altered state.

6.2 Target module

The **Target** module contains logic relevant to deploying the application target that the exploit may be applied to. More specifically, an application target is a specific version of a PHP web application such as WordPress. A copy of the target application with a typical configuration is always loaded before any exploit script is run. For each application, an initialization script is written to provide the **Engine** with the proper details for the installation; 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 have vulnerabilities themselves. 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.

6.3 Exploit class

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 Exploit class is structured such that exploits can be written with concise code functions. Each exploit is simply a subclass of Exploit residing in its own Python file. This class ensures that the Selenium driver is properly initialized and bound to each exploit, and that each exploit script can define actions to take upon installation.

The separation between targets and exploits allows for those unfamiliar with the framework to develop exploits without being concerned with the application or environment setup process, allowing BugBox to be extended without having to learn the intricacies of the entire framework.

Metadata for the exploit module is provided by its attribute dictionary. In it, we declare the name, description, references, target, type, and wiki page for the exploit. The name is the identifier specified in one of the online databases of vulnerabilities mentioned earlier in this paper. The description is a brief statement of what the exploit is supposed to do, because the name is typically does not describe what type of exploit is being applied. The Target is the web application module which is being used for the exploit. The type is the type of exploit being conducted (such as XSS). The wiki page directs a user to a page where auxiliary information about the exploit or script is recorded. The Query module, also part of BugBox, allows for this metadata queried across all exploits (as described in the use cases).

Here is an example body of an exploit procedure used for a simple XSS attack. Scripting the exploit only requires instantiating the Selenium driver and submitting a request 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()
```

6.4 Cleanup and maintenance

Functionality that performs cleanup and maintenance are included in BugBox to ensure that results based on the framework are reproducible. There are many changes that occur in the system over the course of the exploit cycle that will potentially 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.

Since the `chroot` environments on the host machine are represented as directories in the host filesystem, simple scripts are used to maintain them. Scripts using `rsync` are employed to periodically overwrite the `chroot` jails with fresh copies to maintain the integrity of their operating systems. Exploit scripts are also responsible for removing their own side-effects from the target application. For instance, many exploits upload an arbitrary file to be executed by the webserver. The exploit classes can implement a cleanup method which has direct access to the target filesystem, allowing for the environment to be restored to its pre-exploitation state.

7 Lessons learned

In developing BugBox, we learned four key lessons, involving: the need for multiple versions of environments, the utility of visual aids to speed up exploit script development, the importance of representative sampling, and the benefits of building a modular system.

Being able to install web applications onto multiple operating system environments is a critical factor in corpus development. When creating a corpus, a wide variety of examples is required, including web application versions that are outdated. Some of these outdated applications have dependencies on older software versions, which are only included with obsolete Linux distributions. For this reason, we ensured that BugBox supported the use of multiple environments with 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. Being able to visually monitor the execution of the scripts allows developers to efficiently pinpoint errors, streamlining script generation.

Originally, when collecting samples for our corpus, we failed to focus on collecting a representative sample across independent variables such as vulnerability type and age. Having a proper sample is important when evaluating exploit prevention techniques [17] or analyzing source code metrics, to ensure that the result is fair and representative. We will remediate this by collecting new vulnerabilities and stratifying the sample across these variables, and we recommend that future teams collecting vulnerability data apply proper sampling procedures as early in the process as possible.

In our original implementation of this framework, each vulnerability was permanently installed on a separate virtual machine image. We found that introducing new processes which affected the entire corpus (such as beginning to collect execution traces) required large amounts of labor, repeating the process for each VM. The BugBox framework automatically installs vulnerable applications onto a test environment which can be customized at any time, speeding up these tasks. Repetitive research tasks are further facilitated by the use of Python as a layer of abstraction above the target environment, target application, and exploits.

7.1 Ethical Considerations

BugBox is a powerful tool for researchers which could potentially provide a foundation for a fully automated attack platform for malicious attackers. With this in mind, we implemented measures to decrease the potential for malicious use of our system through two rules:

First, no zero-day exploits will be added or accepted to the main distribution of BugBox. Because BugBox is used for research and education, it is sufficient to limit the scope of BugBox to historical vulnerabilities which are already cataloged in any well-known vulnerability or exploit database. Next, all new exploits created or accepted for the framework will only include functionality necessary to test the vulnerability in the BugBox environment and will not contain code that is tailored toward a general-purpose attack. This provides more overhead for an attacker considering using this tool maliciously. Significant modification would be required for an exploit script to carry an actual malicious payload or target vulnerable applications in an unpredictable, real-life environment. Although it is not practical to prevent any user from maliciously using an exploit repository, including our own, by implementing these measures, we seek to limit the audience of malicious users.

8 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 (such as pre-exploit setup requests) and provide a more refined break-down of the exploitation process, as traces would be grouped by HTTP request.

The exploits in the corpus have no standard method for selecting the payload which will 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, serving as a model for implementing this if needed.

More information on obtaining BugBox for research use is available from our website: `http://seam.cs.umd.edu/Projects/BugBox/`. BugBox is designed to work on the Debian GNU/Linux and compatible distributions. The framework can be obtained in the form of a package which can be installed on an existing system. The system 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, and `debootstrap`. Community contributions will be key to building a corpus of sufficient size, therefore, community members may contribute BugBox exploit and target modules as described on the website. The framework is distributed under an open-source license. 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 under contract N000141210147.

References

[1] ANTUNES, N., AND VIEIRA, M. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *Proceedings of PRDC'09: 15th IEEE Pacific Rim International Symposium on Dependable Computing.* (2009), IEEE, pp. 301–306.

[2] AUSTIN, A., AND WILLIAMS, L. One technique is not enough: A comparison of vulnerability discovery techniques. In *Proceedings of Empirical Software Engineering and Measurement (ESEM), International Symposium on* (Sept 2011), pp. 97 –106.

[3] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Theory and techniques for automatic generation of vulnerability-based signatures. *Dependable and Secure Computing, IEEE Transactions on 5*, 4 (Oct-Dec 2008), 224 –241.

[4] D'AMBROS, M., LANZA, M., AND ROBBES, R. An extensive comparison of bug prediction approaches. In *Proceedings of Mining Software Repositories (MSR), 7th IEEE Working Conference on* (2010), IEEE, pp. 31–41.

[5] FONSECA, J., AND VIEIRA, M. Mapping software faults with web security vulnerabilities. In *Proceedings of IEEE DSN '08 Conference on Dependable Systems and Networks* (2008), pp. 257 –266.

[6] Huynh, T., and Miller, J. An empirical investigation into open source web applications implementation vulnerabilities. *Empirical Software Engineering 15*, 5 (2010), 556–576.

[7] Ku, K., Hart, T. E., Chechik, M., and Lie, D. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2007), ASE '07, ACM, pp. 389–392.

[8] Livshits, B. Defining a set of common benchmarks for web application security. In *Workshop on Defining the State of the Art in Software Security Tools* (Baltimore, August 2005), Citeseer.

[9] NIST. Samate reference dataset. `http://samate.nist.gov/SRD/`.

[10] NIST. National vulnerability database. `http://nvd.nist.gov/`, 2013.

[11] Offensive Security. Exploit db. `http://www.exploit-db.com/`.

[12] Offensive Security. Osvdb: The open source vulnerability database. `http://www.osvdb.org/`, 2013.

[13] OWASP. Webgoat. `https://code.google.com/p/webgoat/`, 2013.

[14] Rapid7. Metasploit framework. `http://www.metasploit.com/`, 2013.

[15] Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on 37*, 6 (2011), 772–787.

[16] Stuckman, J., and Purtilo, J. Analyzing the wikisphere: Methodology and data to support quantitative wiki research. *Journal of the American Society for Information Science and Technology 62*, 8 (2011), 1564–1576.

[17] Stuckman, J., and Purtilo, J. A testbed for the evaluation of web intrusion prevention systems. In *Proceedings of MetriSec: International Workshop on Security Measurements and Metrics* (Sept 2011), IEEE, pp. 66 –75.

[18] Stuckman, J., and Purtilo, J. Comparing and applying attack surface metrics. In *Proceedings of MetriSec: International Workshop on Security Measurements and Metrics* (2012).

[19] Su, Z., and Wassermann, G. The essence of command injection attacks in web applications. In *Conference record of POPL '06: The 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 372–382.

[20] WhiteHat. Website security statistic report 9th edition. `https://www.whitehatsec.com/assets/WPstats_spring10_9th.pdf`, 2010.

[21] Williams, C., and Hollingsworth, J. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on 31*, 6 (June 2005), 466 – 480.

[22] Zimmermann, T., Premraj, R., and Zeller, A. Predicting defects for eclipse. In *Proceedings of PROMISE'07: ICSE Workshop on Predictor Models in Software Engineering* (may 2007), p. 9.