BugBox : A Vulnerability Corpus for PHP Web Applications

Gary Nilson
Computer Science Department
University of Maryland

Kent Wills
Computer Science Department
University of Maryland

Jeff Stuckman
Computer Science Department
University of Maryland

Jim Purtilo
Computer Science Department
University of Maryland

1 TODO

Our previous approach

Abstract

Web Applications are a rich source of vulnerabilities due to their high exposure, ease of exploitation, and pervasive poor programming practices. Accordingly, web applications are ideal specimens for *empirical vulnerability research*. There is currently no publicly available resource of vulnerabilities that is both robust and practical to use. BugBox is an open-source corpus and framework for PHP web application vulnerabilities that was created to fill this void with the goal of encouraging greater quality security research. In this paper we discuss the use of the framework and further explain the motivating factors behind the design. The corpus can be used to enhance testing vulnerability indicators and metrics, developing vulnerability definition representations, testing intrusion detection systems, or for creating demonstrations for training purposes.

2 Introduction

Web applications are subject to a rich variety of exploit types, such as cross-site scripting (XSS), cross-site request forgery (XSRF), buffer overflow, and SQL injection. A recent study by White Hat Security [**?**] analyzed seven web application languages: ASP, ASPX, CFM, DO, JSP, PHP, and PL 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 and preponderance 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 exist and disclose the details of the vulnerability and/or exploit, they do not provide a collection of structured vulnerable code that can be used for analysis. Specifically, it is not enough to know what exploit was used in a situation, but we must also know the location of the vulnerable code that allowed the exploit to execute. A suitable, structured collection of vulnerable code is typically not made publicly available and is scarce in quantity. The structured collection, often reffered to as a corpus, is useful in statistical analysis and hypothesis testing, but only if it is of certain quality. To elaborate one must consider several criteria in order to determine the quality or suitability of a corpus: representativeness of the samples, quantity of the samples, diversity of the samples, and reproducability of the tests that produced the samples [need ref].

A suitable corpus of this type can aid work in a wide variety of applications, such as: evaluating static analysis tools [**?**], penetration testing, training security teams with vulnerability injection [**?**], conducting analysis on open source web applications [**?**], and as a way to compare attack surface metrics accross applications [**?**].

Understanding the scarcity and labor involved in developing a suitable corpus has led to the in-house creation of BugBox, a publicly available, scalable, efficient, easy-to-use framework and corpus that collects vulnerable code signatures through an automated trace-based collection method. Our goal is to expand and diversify the content of BugBox through community support.

NEEDS REVISION, less MBA mumbo jumbo Support that will be effortlessly streamlined because of a scalable framework that is easy to maintain, quick to use, automated, and fully compartmentalized. Furthermore, the framework capability is streamlined through the thorough use of abstraction and encapsulation mak-

ing the management of the application, environment, and exploitation in BugBox.

The design is intended to make it practical to manage a large database of exploits, along with their target environments while acting as more than an exploit repository through containing the software and its dependencies in which a vulnerability exists, the configuration of the software in it's vulnerable state, exploit code that will trigger the security breach, and data that describes any distinguishing attributes of the bug.

3 Vulnerability Representation

Line-based, run-based, and trace-based approaches are all effective ways for representing the *vulnerability point*[ref] in a program. The study of identifying vulnerabiliyt points and their representations is often refered to as *vulnerability localization*[ref]. A line-based[**?**] 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[**?**] 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. The method proposed for our corpus is a trace-based collection approach in order to capture code vulnerabilities in software. We accomplish this task by collecting execution traces with XDebug showing the exploit and then manually highlighting the program paths relevant to the vulnerability.

4 Anatomy of a Corpus Entry

A corpus entry has three main entities: a script, a trace, and a package. The script contains the exploit actions that are programmatically taken on a specific application. This script helps automation and reproducibility of results. The trace is the resultant execution trace of the web application when the script is executed on it. The trace collected is why we have identified our method as a trace-based collection approach. The packages represent web applications in their vulnerable state. Each web application is represented multiple times under different versions, giving accessability to the vulnerable code.

5 Corpus Entry Workflow

There are a few simple steps to creating a new corpus entry: determine the exploit, download a package, confirm the exploit, create the script, collect the trace. As mentioned in the introduction, one could use any vulnerability or exploit database. Once a particular exploit is found, we verify if the package is available already in the corpus, if not we simply upload it as a new package. We perform a manual execution of the exploit to verify that we understand the scope of the exploit, from which we proceed to write a script for the exploit so that the process will be automated for future use. Finally, we collect the trace of the program while our script is being executed and add it to our corpus.

6 Usage

The following Python script illustrates a use-case for the BugBox framework. In this example, all XSS vulnerabilities existing in the the corpus are sequentially invoked, and execution traces are collected:

```
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.exploit()
    engine.xdebug_autotrace_off()
    engine.shutdown()
```

This job would have taken hours to accomplish using our previous approach, which was based on virtual machine snapshots. With a layer of abstraction between the user and the OS, target application, and exploits, it is now possible to automate research tasks easily. The following sections will give details on the system design that makes this automation possible.

TRANSITION or elsewhere Many times the focus of study is on a single vulnerability or vulnerable application. The run-time management of single corpus entries can be done easily 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 <disp_on | disp_off>>
    stop        <exploit_name>
    trace_on    <exploit_name>
    trace_off   <exploit_name>
    autorun     <exploit_name>
```