TESTREX: a Testbed for Repeatable Exploits

Stanislav Dashevskyi
Security & Trust, FBK-Irst
DISI, University of Trento

Daniel Ricardo dos Santos
Security & Trust, FBK-Irst
DISI, University of Trento

Fabio Massacci
DISI, University of Trento

Antonino Sabetta
SAP Labs France

Abstract

Web applications are the target of many known exploits and also a fertile ground for the discovery of security vulnerabilities. Those applications may be exploitable not only because of the vulnerabilities in their source code, but also because of the environments on which they are deployed and run. Execution environments usually consist of application servers, databases and other supporting applications. In order to test whether known exploits can be reproduced in different settings, better understand their effects and facilitate the discovery of new vulnerabilities, we need to have a reliable testbed. In this paper, we present TESTREX, a testbed for repeatable exploits, which has as main features: packing and running applications with their environments; injecting exploits and monitoring their success; and generating security reports. We also provide a corpus of example applications, taken from related works or implemented by us.

1 Introduction

Vulnerable web applications are one of today's targets [19] that are very hard to identify by traditional black-box approaches for security testing [13, 6, 21]. Industry approaches to black-box application security testing (e.g., IBM AppScan) or academic ones (e.g., Secubat [11] or BugBox [15]) require security researchers to write down a number of "general-purpose" exploits that can demonstrate the (un)desired behavior. Such exploits can then be tested on one's application of choice.

As a matter of fact, web applications can be deployed and run in many different execution environments, consisting of operating systems, application servers, database servers and other sorts of supporting applications in the backend, as well as different configurations in the frontend [13]. Two illustrative examples are SQL injection exploits (which depend on the capabilities of the underlying database and the authorizations of the user who runs it [20]), and XSS exploits (which depend on the browser being used and its rules for executing or blocking JavaScript code [22]). These different environments may transform failed attempts into successful exploits and vice versa. The information about the configuration is an intrinsic part of the vulnerability description. Since the operating system and supporting applications in the environment can also have different versions, this easily escalates to a huge number of combinations which can be hard to manually deploy and test.

If we want to experiment with web-application security testing, then building a corpus of vulnerabilities and exploits (e.g., BugBox [15] or WebGoat [18]) is just the first step because each "vulnerability" might be successfully exploited only for the particular configuration used to build the corpus. We also need a way to automatically switch configurations and re-test the exploit to check whether they worked with a different configuration. Such data should also be automatically collected so that a researcher can see how different exploits work once the configuration changes. Such automatic process of "set-up configuration, run exploit, measure result" was proposed by Allodi et al. [1] for testing exploit kits but it is not available for testing web-applications.

Our proposed solution, TESTREX[1], combines packing applications and execution environments that can be easily and rapidly deployed, scripted exploits that can be automatically injected, useful reporting and an isolation between running instances to provide a real "playground" and experimental setup where security testers and researchers can run their tests and experiments and get reports at various levels of detail.

We also provide a corpus of vulnerable web applications to illustrate the usage of TESTREX over a variety of web programming languages. The vulnerability corpus is summarized in Table 1. Some of them are taken from other sources (e.g., BugBox [15] and WebGoat [18]),

[1] http://securitylab.disi.unitn.it/doku.php?id=testrex

while others are developed by us. For the latter category, we focused on server-side JavaScript (JS), because of its growing popularity in both open-source and industrial usage (e.g., Node.js and SAP HANA) and, to the best of our knowledge, the lack of vulnerability benchmarks. We are currently extending TESTREX to cover also SecuriBench [12].

Source	Language	Exploits
BugBox [15]	PHP	83
WebGoat [18]	Java	10
Our examples	Server-side JS	7

Table 1: Summary of the exploits in the corpus

This paper is organized as follows: Section 2 introduces related work while Section 3 presents an overview of the proposed testbed; Section 4 details the exploits and vulnerabilities that we used; Section 5 discusses implementation details; Section 6 shows some usage examples; Section 7 discusses the lessons learned; Section 8 presents our ideas for applying the testbed in an industrial setting; finally, Section 9 is the conclusion.

2 Related work

Empirical security research has been recognized as very important in recent years [7, 14, 5]. However, a number of issues should be tackled in order to correctly implement it. These issues include isolation of the experimental environment [3, 1, 4, 15], repeatability of individual experiments [7, 1], collection of the experimental results, and justification of the collected data [14].

The use of a structured testbed can help in achieving greater control over the execution environment, isolation among experiments and reproducibility. However, most proposals for security research testbeds focus on the network level (e.g., DETER [3], ViSe [2] and vGrounds [9]).

On the application level there are significantly less experimental frameworks. The BugBox [15] framework is one of them. It provides the infrastructure for deploying vulnerable PHP-MySQL web applications, creating exploits and running these exploits against applications in an isolated and easily customizable environment. As in BugBox, we use the execution isolation and environment flexibility concepts. However, we needed to have more variety in software configurations and process those configurations automatically. We have broaden the configurations scope by implementing software containers for different kinds of web applications, and automatically deploy them along the line of the MalwareLab by Allodi et al. [1].

The idea of automatically loading a series of clean configurations every time before an exploit is launched was also proposed by Allodi et al. in their MalwareLab [1]. They load snapshots of virtual machines that contain clean software environment and then "spoil" the environment by running exploit kits. This eliminates the undesired cross-influence between separate experiments and enforces repeatability. So we have incorporated it into TESTREX. For certain scenarios cross-influence might be a desired behavior, therefore TESTREX makes it possible to run an experiment suite in which the experimenter can choose to start from a clean environment for each individual exploit/configuration pair or to reuse the same environment for a group of related exploits.

Maxion and Killourhy [14] have shown the importance of comparative experiments for software security. It is not enough to just collect the data once, it is also important to have the possibility to assess the results of the experiment. Therefore TESTREX includes functionalities for automatically collecting raw statistics on successes and failures of exploits.

3 Overview

The testbed should help security researchers to answer (semi) automatically a number of security questions. Given an exploit X that successfully subverts an application A running on an environment E:

1. Will X be successful on application A running on a new environment E'?
2. Will X be successful on a new version of the application, A', running on environment E?
3. Will X also be successful on a new version of the application, A', running on a new environment E'?

These questions can be exemplified in the following situation:

Example 1 *We have a working SQL injection exploit for WordPress 3.2 running with MySQL and we would like to know whether (i) the same exploit works for WordPress 3.2 running with PostgreSQL; (ii) the same exploit works for WordPress 3.3 running with MySQL; and (iii) the very exploit works for WordPress 3.3 and PostgreSQL.*

We use this example throughout the paper to illustrate the concepts and components used in the testbed.

The main inner loop of the testbed is the obvious to run an experimental suite and namely:

1. Pack and extract applications with their environments
2. Deploy an application in a suitable configuration
3. Run the application
4. Inject an exploit in the running application
5. Identify the success or the failure of the exploit
6. Report the result and store the execution log

The process can be performed manually, without injecting automated exploits, or can be performed automatically by running several applications and exploits in a batch.

One of the main targets of TESTREX is to make this process as automatizable as possible. Another important feature, that we already mentioned, is that the execution of the application and the exploit can happen in an isolated and clean environment. In this way every test is run on a clean slate. A side benefit of this testing mode is that different tests can be run in parallel.

Figure 1 shows an overview of the testbed architecture. The main component is the Execution Engine, which takes as input a *Configuration* and an *Exploit* and outputs a Report. The inputs listed in the Figure 1 are further detailed

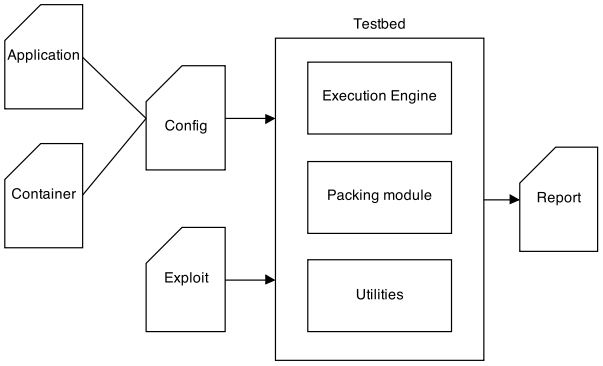

Figure 1: Overview of the testbed architecture

below:

- An *Application* is simply a set of files containing the code of the system under test.
- A *Container* is the representation of the execution environment. It is an image of the system on which the application must be run, containing an operating system and supporting applications, like an application server and a database management system.
- A *Configuration* is a set of files used to bind an *Application* to a *Container*. It describes the setup required for a given application to run in a given container, like preloading a database, creating users and starting a server.
- An *Exploit* is a sequence of steps that must be taken in order to cause unintended behavior, taking advantage of a vulnerability in the application [8].
- A *Report* contains as main information a result (success/fail) of the execution of an exploit on a configuration. Metadata about the exploit and logs of its execution can be attached to the report, in order to provide further details that allow, in a closer manual inspection, to determine why an exploit was not successful in a given environment.

The tester then chooses one configuration against which the exploit is going to be run. The Execution Engine is invoked with the chosen configuration and the exploit as inputs and outputs the result of the execution. It builds and loads an execution environment (application and its container), injects the exploit in the running application, monitors the success of the exploit execution and later destroys the loaded environment (if a clean-slate approach is sought).

TESTREX also includes some additional utilities. The Packing Module allows testers to package the applications and execution environments in a compressed archive file that can be easily deployed in another system running the testbed. The Utilities are a collection of scripts to import applications and exploits from other sources, such as BugBox, and to manage the containers.

Example 2 *The inputs for Example 1 are instantiated as follows*

Application: *There are two, each one is a set of* `.html`*,* `.php` *and* `.js` *files in a* `Wordpress` *folder.*
Container: *There are two, one with Ubuntu, Apache and MySQL and one with Ubuntu, Apache and PostgreSQL.*
Configuration: *There are four, one for WP3.2 with MySQL, one for WP3.3 with MySQL, one for WP3.2 with PostgreSQL and one for WP3.3 with PostgreSQL.*
Exploit: *There is only one, it is a script that navigates to the vulnerable page, interacts with it and injects a payload, simulating the actions of an attacker.*

A key requirement was that the architecture should be easily extensible to allow for the inclusion of new exploits, applications and execution environments. To this extent we have organized exploits in classes and tried to use some of the properties of inheritance hierarchies (we detail this in the next Section).

4 Exploits

In our setting, exploits are unit tests: (1) every exploit is self-contained and can be executed independently; and (2) every exploit is targeted to take advantage of a certain vulnerability in a certain application.

When using the testbed in a specific application, the exploit can be written by the tester or taken from a public source, but in any case, it must be compliant with what we expect from an exploit (detailed in Section 5).

In order to verify the applicability of our testbed, we have taken the WebGoat vulnerable application [18] and developed 10 example exploits for it. We have also developed exploits for 7 specially crafted vulnerable applications, to demonstrate SQL injection, NoSQL injection, stored and reflected XSS, path traversal and code injection vulnerabilities in Node.js [10] applications. The

path traversal and the code injection examples take advantage of recently discovered vulnerabilities in Node.js modules [16, 17].

The testbed also supports the possibility of importing applications and exploits from BugBox by running a single script on a BugBox package. The script copies the applications and exploits into the corresponding folders under the testbed and creates identical configuration files for every application, using Apache as a web server and MySQL as a database server. We are able to run most of the BugBox native exploits and collect statistics without modifying their source code.

Table 2 shows detailed information on all of the exploits that we have implemented. In the table, *Source* indicates where we took the exploits from, *Language* is the language in which the vulnerable applications are implemented, *Applications* lists the application taken from that source, *Containers* shows which containers were used to run the applications and *Exploits* is the number of exploits of each type that are successfully run.

5 Implementation

The testbed is implemented in Python, mainly because of the possibility of fast and easy prototyping and integration with other technologies, such as Docker and Selenium. In the next subsections, we describe in details the implementation of each component of the testbed.

5.1 Execution Engine

The Execution Engine is the main Python module that binds all the modules and features together. It supports three modes of operation: *single*, *batch* and *manual* runs.

The single mode allows testers to specify and run a desired exploit against a chosen application just once. This is useful if a tester wants to quickly check whether the same exploit works for a few different applications, different versions of the same application or the same application in different environments. A `.csv` report is generated at the end of the run.

To run applications and exploits in the batch mode, we loop through a folder containing exploit files and inject them in their respective configurations, generating a summary `.csv` report in the end. In this mode, the Execution Engine maps exploits to applications by scanning the metadata in each exploit for the appropriate targets.

To manually test the applications, we simply stop the Execution Engine when the environment is built and loaded, and return to the tester a browser that he/she can use to test the application in a safe environment. No report is generated in this case.

5.2 Applications

Applications are packaged as `.zip` files containing all their necessary code and supporting files, such as database dumps.

Unpacked applications must be located under the `<testbed_root>/data/targets/applications` folder to be accessible by the Execution Engine.

As an example, we provide some applications with known vulnerabilities, presented in Table 2. Most of them are known real-world applications, but there are also some small examples developed to explore issues in server-side JavaScript applications. We developed these examples because we are unaware of known benchmarks or vulnerable examples of this kind of applications.

5.3 Containers

Instead of creating virtual machines for the applications and their software configurations, we employ Linux Containers[2], which is a technology that provides virtualization capabilities on the operating system level. Containers are sandboxed filesystems that reuse the same Linux kernel, but have no access to the actual operating system where they are deployed.

Docker[3] provides a format for packing, shipping and running applications with a lightweight file repository, all on top of Linux Containers. We use Docker to implement our two types of containers, *software-specific* and *application-specific*, and build our testbed on top of these, with scripts to build and control the containers. However, Docker has no means to automatically run our exploits on a desired system.

Downloading generic software components and building a Docker container every time an application has to be run might be resouce- and time-consuming. Therefore, we maintain software-specific containers that consist of generic software components required for certain types of web applications. Such containers encapsulate operating system, server and database engine, and have to be built only once.

These software-specific containers are described in Dockerfiles (a format defined by the Docker project), named as: `<operating_system>-<webserver>-<database>-<others>`. We provide some predefined containers for common environments, such as:

ubuntu-apache-mysql A classic LAMP server, containing the Ubuntu distro, the Apache web server, the MySQL database and the PHP programming language;

ubuntu-node-mongo Containing the Ubuntu distro, the Node.js web server and the MongoDB database;

[2] https://linuxcontainers.org/

[3] https://www.docker.io/

Source	Language	Applications	Containers	Exploits
BugBox	PHP	WordPress, CuteFlow, Horde, PHP Address Book, Drupal, Proplayer, Family Connections, AjaXplorer, Gigpress, Relevanssi, PhotoSmash, WP DS FAQ, SH Slideshow, yolink search, CMS Tree page view, TinyCMS, Store Locator Plus, phpAccounts, Schreikasten, eXtplorer, Glossword, Pretty Link	ubuntu-apache-mysql	XSS (46), SQLi (17), Code Execution (7), Authentication Bypass (4), Information Disclosure (2), LFI (2), CSRF (2), Denial of Service (1)
WebGoat	Java	WebGoat	ubuntu-tomcat-java	SQLi (2), XSS (2), Authentication Flaws (3), Database Backdoor (1), Parameter Tampering (2)
Our examples	Server JS	CoreApp, JS-YAML, NoSQLInjection, ODataApp, SQLInjection, ST, WordPress3.2, XSSReflected, XSSStored	ubuntu-node, ubuntu-node-mongo, ubuntu-node-mysql	XSS (3), NoSQLi (1), SQLi (1), Path traversal (1), Code Injection (1)

Table 2: Details of the exploits in the corpus

ubuntu-node-mysql Containing the Ubuntu distro, the Node.js web server and the MySQL database.

Application-specific containers are built on top of software-specific containers every time the Execution Engine runs an application. The Execution Engine clones a corresponding software-specific container and adds the application files to the clone - building a new container this way is just a matter of seconds. When the Execution Engine finishes the run, the used container is deleted to free disk space.

5.4 Configurations

Every application must have a set of configuration files that specify how to deploy and set up the application within the corresponding application-specific container.

Each configuration consists of at least two files: the Dockerfile that is used to build the application's container and a shell script file with additional commands that must be executed within the container (like running a server instance or starting a database server). This file is usually called `run.sh`.

The configuration files must be placed in a separate folder under the configurations root folder (`<testbed_root>/data/targets/configurations`). We use certain naming conventions to make it possible for the Execution Engine to match applications with corresponding configuration files: `<app-name>__<app-container-name>`.

Example 3 *A configuration folder for the application `Wordpress_3.2`, might have the names `Wordpress_3.2__ubuntu-apache-mysql` or `Wordpress_3.2__ubuntu-apache-postgresql`, depending on the container used for it.*

Listings 1 and 2 present an example of a Dockerfile and a `run.sh` file, used to configure a WordPress application in the `ubuntu-apache-mysql` container.

In Listing 1, line 1 specifies that the container for this application is built on top of the `ubuntu-apache-mysql` container. In lines 2 and 3, the application is imported to the `/var/www/wordpress` folder in the container and in lines 4 and 5, the `run.sh` script is invoked inside the container.

```
FROM ubuntu-apache-mysql
RUN mkdir /var/www/wordpress
ADD . /var/www/wordpress
RUN chmod +x /var/www/wordpress/run.sh
CMD cd /var/www/wordpress && ./run.sh
```

Listing 1: Dockerfile example

```
#!/bin/bash
mysqld_safe &
sleep 5
mysql < database.sql
mysqladmin -u root password toor
apache2ctl start
```

Listing 2: Shell script file example

In Listing 2, lines 2-5 are used to start the database server and preload application data. Line 6 starts the Apache web server.

5.5 Exploits

The exploits are implemented as Python classes that share common properties: (1) every exploit contains metadata describing its characteristics such as name, description, type, target application and container; (2) logging and reporting capabilities - exploit classes maintain logging information and results of the run, passing this information to the Execution Engine.

The *Selenium Web Driver*[4] automates web browsers, supporting visualization, JavaScript execution and DOM interaction [15]. BugBox uses Selenium and, since the mentioned features are important for our work, we also used it to create our own exploits.

[4] http://docs.seleniumhq.org/projects/webdriver/

Every Selenium-based exploit in the testbed is subclassed from the `BasicExploit` class, which encapsulates generic functionality: the way Selenium is used to automate the web browser, `setUp()` and `tearDown()` routines, logging and reporting, etc.

In order to create a new exploit, the tester has to create a new exploit class, specify the exploit-specific metadata and override the `runExploit()` method by adding a set of actions required to perform an exploit.

Verifying the success of an exploit is also done within the `runExploit()` method - differently for every exploit. This allows us to handle complex exploits that are not always repeatable, such as heap spraying. For such cases, the exploit can be specified to run a certain number of times until it is considered a success or a failure.

5.6 Report

A report is a `.csv` file that the Execution Engine creates or updates every time it runs an exploit. Every report contains one line per exploit that was executed. This line consists of: names of the exploit and the target application, application-specific container, type of the exploit, the exploit start-up status and the overall result and other data. Along with this report the Execution Engine maintains a log file that contains information which can be used to debug exploits.

Example 4 *The listing below shows a single entry from the* Wordpress_3_2_XSS *exploit that was run against the* WordPress 3.2 *application.*

```
Wordpress_3_2_XSS, Wordpress3.2, ubuntu-apache-mysql,
    XSS, CLEAN, SUCCESS, SUCCESS, 30.345, Exploits
    for "XSS vulnerability in WordPress application"
```

Listing 3: An example of the report file entry after the exploit run

6 Example usage

In this Section, we describe in details the steps needed to add an experiment to the testbed, given an existing application. The steps consist of: adding the application; creating the configuration files; building containers; creating and running the exploits. Again we use WordPress 3.2 as the example application.

6.1 Deploying the application

The code of the application must be copied into a separate folder under the applications root "`<testbed_root>/data/targets/applications`". The folder name must correspond to a chosen name of the application in the testbed.

To deploy the WordPress 3.2 application, copy all of its files to the folder "`<testbed_root>/data/targets/applications/WordPress_3_2`".

6.2 Creating configuration files and building containers

Due to our naming conventions, the name of the configuration folder must be the same as the name of the corresponding Docker image (please see Section 5.3).

If there is no software-specific container that might be reused by the application, this container must be created in the first place. Configuration files for software-specific containers are located under the "`<testbed_root>/data/targets/containers`" folder.

In our example, we create a software-specific container with the `ubuntu-apache-mysql` name, since the application requires *Apache* as a web server and *MySQL* as a database engine. To do this, we create a Dockerfile under `<testbed_root>/data/targets/containers/ubuntu-apache-mysql` that contains the code shown in Listing 4 and build it with the script in `<testbed_root>/util/build-images.py`.

```
FROM ubuntu:raring
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get -y install
      mysql-client mysql-server apache2 libapache2-mod
      -php5 php5-mysql php5-ldap
RUN chown -R www-data:www-data /var/www/
EXPOSE 80 3306
CMD ["mysqld"]
```

Listing 4: The Dockerfile for building the `ubuntu-apache-mysql` software-specific container

As a next step, we create the configuration files for the application-specific container. We create a Dockerfile and a shell script file under the "`<testbed_root>/data/targets/configurations/Wordpress_3_2__ubuntu-apache-mysql`" folder (please see Section 5.4 for the explanation and Listings 1 and 2 for the code examples).

There is no need to build this container, since it is done by the Execution Engine for every run.

6.3 Creating and running the exploit

Finally, we create an exploit for the Wordpress 3.2 application by creating a file with a Python class under the "`<testbed_root>/data/exploits`" folder. The new exploit class must be subclassed from the already existing `BasicExploit` class. As a last step, we specify the exploit's metadata in the `attributes` dictionary and put the exploit steps into the `runExploit()` method. Listing 5 shows the exploit class.

The `attributes` dictionary contains exploit metadata and the `runExploit()` method contains the steps required

to reproduce the exploit and to check its success. Listing 6 shows the list of commands available to run the newly created application with the Execution Engine.

```
from BasicExploit import BasicExploit
class Exploit(BasicExploit):
  attributes = {
      'Name' :          'Wordpress_3_2_XSS',
      'Description' : "XSS attack in Wordpress
           application",
      'Target' :        "Wordpress3.2",
      'Container': 'ubuntu-apache-mysql',
      'Type' : 'XSS'
   }

   def runExploit(self):
      wp = self.wrapper
      wp.navigate("http://localhost:49160/wordpress/
           wp-admin/post-new.php?post_type=page")
      [...]
      self.assertIn("XSS", alert_text, "XSS")
```

Listing 5: Wordpress_3_2_Exploit.py file contents

```
#1: Single mode
./run.py --target Wordpress_3_2__ubuntu-apache-mysql
    --exploit Wordpress_3_2_Exploit.py

#2: Batch mode
./run.py

#3: Batch mode only for Wordpress3.2 app
./run.py --target Wordpress_3_2__ubuntu-apache-mysql

#4: Manual mode
./run.py --manual WordPress_3_2__ubuntu-apache-mysql
```

Listing 6: Running exploits against the *WordPress 3.2* application

By default, the execution report is saved into the "`<testbed_root>/reports/ExploitResults.csv`" file. In order to specify a different location for the results, a tester has to add additional parameter to the `run` command: `--results new/location/path.csv`.

7 Lessons Learned

During the design and development of TESTREX, the key lessons learned were: the value of building on top of existing approaches; the importance of having a simple and modular architecture; and the necessity of reliable information on applications, exploits and execution environments.

Building on top of the related work, like we did with BugBox for the format of our exploits and MalwareLab for the design of our experiments, was extremely valuable. This reduced our design and development time, and allowed us to quickly have a large corpus of applications and exploits on which we could test our work.

Having a simple architecture and being able to add small modules to it was crucial in the development of TESTREX, because this way we could test many options for the supporting frameworks we used (like Selenium and Docker) and select those that best fit our purposes.

When adding the experiments to the TESTREX, we soon learned that the hardest part is writing the configuration files that allow an application to run on top of a container. This is because the information on how to configure an application for a certain environment usually is not detailed enough. Also, many times the descriptions of exploits that are available are vague, limited to a proof of concept or unreliable.

8 Industrial Usage

There are several uses of TESTREX that we are exploring in an industrial setting, covering different phases of the software development lifecycle and fulfilling the needs of different stakeholders. In the following we summarize the directions that we deem more promising.

"Executable documentation" of vulnerability findings. When a vulnerability is found in a product, being able to reproduce an attack is key to investigate the root cause of the issue and to provide a timely solution. It is current practice to use a combination of natural language and scripting to describe the process and the configuration necessary to reproduce an attack. The results of which are erratic, complicating the task of the security response department.

TESTREX exploit scripts and configurations can be thought of as "executable descriptions" of an attack. The production of exploits and configurations could not just be the task of the security validation department, but also of external security researchers, for which the company might set up a bounty program requiring that vulnerabilities are reported in the form of TESTREX scripts.

Automated validation and regression testing. As part of the software development lifecycle, TESTREX can be used to check the absence of known vulnerabilities or to perform regression tests to verify that a previously fixed vulnerability is not introduced again.

To this end, a corpus of exploits and configurations is stored in a corporate-wide repository and is used to perform automated tests all along the development cycle. In large corporations, the results of these tests are part of the evidence needed in order to pass quality assurance gates. Currently, much of the process to produce such evidence relies on manual work, which increases cost, errors and unpredictability of the process. TESTREX can be used to accelerate and improve the effectiveness and the predictability of quality assurance processes.

Support for penetration testing. An important problem arising in pen-testing large systems is the complexity of setting-up and reproducing the conditions of the target system – typically involving many hosts and software components, each of which may need to be configured in a specific way. A key strength of our framework is the ability to capture these configurations as reusable scripts;

this requires a non-negligible effort, but the results can be reused across different pen-testing sessions. This has the advantage of providing automation, reproducibility, and the ability to proceed stepwise in the exploration of the effect of different configurations and versions of the software elements on the presence (or absence) of vulnerabilities in the system.

9 Conclusion and Future work

In this paper, we presented TESTREX, a Testbed for Repeatable Exploits that combines a way of packing applications and execution environments, automatic execution of scripted exploits, and reporting to provide an experimental setup for security testers and researchers. We also provided a corpus of applications and exploits, taken from related works or developed by us, to show the range of applications that can be handled by TESTREX.

Besides expanding our corpus, we intend to apply TESTREX for several research activities, such as large-scale testing of static analysis tools and semi-automatic generation of test cases for web applications.

To move towards the generation of test cases, we will refine our implementation of exploits into a hierarchy of exploit classes. For instance, a WordPress SQLi exploit will extend `SQLInjectionExploit`, subclassed from `BasicExploit`. This will help to write exploits faster, by factoring common attributes of exploit types and altering the exploit attributes in case if a given exploit did not work.

Another possibility of future development is helping testers in finding minimum necessary environment sets required for an exploit to succeed against an application.

Acknowledgments. The work of the University of Trento was partly supported by the EU under grants FP7-ICT-NESSOS, FP7-PEOPLE-SECENTIS, and FP7-SEC-SECONOMICS, and the Italian MIUR under grant PRIN-TENACE.

References

[1] ALLODI, L., KOTOV, V., AND MASSACCI, F. Malwarelab: Experimentation with cybercrime attack tools. *Proc. of CSET 13* (2013).

[2] ARNES, A., HAAS, P., VIGNA, G., AND KEMMERER, R. Digital forensic reconstruction and the virtual security testbed vise. In *Detection of Intrusions and Malware & Vulnerability Assessment*, R. Bschkes and P. Laskov, Eds., vol. 4064 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 144–163.

[3] BENZEL, T. The science of cyber security experimentation: The deter project. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 137–148.

[4] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., GUIZANI, W., KACZMAREK, M., MARION, J.-Y., ST-ONGE, P.-L., ET AL. Isolated virtualised clusters: testbeds for high-risk security experimentation and training. In *Proceedings of the 3rd international conference on Cyber security experimentation and test (Berkeley, CA, USA, 2010), CSET* (2010), vol. 10, pp. 1–8.

[5] CARROLL, T. E., MANZ, D., EDGAR, T., AND GREITZER, F. L. Realizing scientific methods for cyber security. In *Proceedings of the 2012 Workshop on Learning from Authoritative Security Experiment Results* (New York, NY, USA, 2012), LASER '12, ACM, pp. 19–24.

[6] CURPHEY, M., AND ARAWO, R. Web application security assessment tools. *Security Privacy, IEEE 4*, 4 (July 2006), 32–41.

[7] EIDE, E. Toward replayable research in networking and systems. *Position paper presented at Archive* (2010).

[8] FONG, E., GAUCHER, R., OKUN, V., BLACK, P. E., AND DALCI, E. Building a test suite for web application scanners. *2014 47th Hawaii International Conference on System Sciences 0* (2008), 479.

[9] JIANG, X., XU, D., WANG, H., AND SPAFFORD, E. Virtual playgrounds for worm behavior investigation. In *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds., vol. 3858 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 1–21.

[10] JOYENT, INC. Node.js. `http://nodejs.org/`, 2014.

[11] KALS, S., KIRDA, E., KRGEL, C., AND JOVANOVIC, N. Secubat: a web vulnerability scanner. In *WWW* (2006), pp. 247–256.

[12] 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* (August 2005).

[13] LUCCA, G. A. D., AND FASOLINO, A. R. Testing web-based applications: The state of the art and future trends. *Information and Software Technology 48*, 12 (2006), 1172 – 1186. Quality Assurance and Testing of Web-Based Applications.

[14] MAXION, R. A., AND KILLOURHY, K. S. Should security researchers experiment more and draw more inferences? In *Workshop on Cyber Security Experimentation and Testing* (August 2011), CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[15] NILSON, G., WILLS, K., STUCKMAN, J., AND PURTILO, J. Bugbox: A vulnerability corpus for php web applications. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test* (Berkeley, CA, 2013), USENIX.

[16] NODE SECURITY PROJECT. Js-yaml deserialization code execution. `https://nodesecurity.io/advisories/JS-YAML_Deserialization_Code_Execution`, 2013.

[17] NODE SECURITY PROJECT. st directory traversal. `https://nodesecurity.io/advisories/st_directory_traversal`, 2014.

[18] OWASP. Webgoat. `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`.

[19] SCHOLTE, T., BALZAROTTI, D., AND KIRDA, E. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security* (Berlin, Heidelberg, 2012), FC'11, Springer-Verlag, pp. 284–298.

[20] STUTTARD, D., AND PINTO, M. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[21] TRIPP, O., FERRARA, P., AND PISTOIA, M. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *International Symposium on Software Testing and Analysis* (2014).

[22] ZALEWSKI, M. *The Tangled Web: A Guide to Securing Modern Web Applications*, 1st ed. No Starch Press, San Francisco, CA, USA, 2011.