**TestREx: a Framework for Repeatable Exploits**

Stanislav Dashevskyi¹, Daniel Ricardo dos Santos¹,², Fabio Massacci¹, and Antonino Sabetta³

¹ University of Trento, Italy stanislav.dashevskyi@unitn.it, fabio.massacci@unitn.it
² Fondazione Bruno Kessler, Italy dossantos@fbk.eu
³ SAP Labs, France antonino.sabetta@sap.com

This draft was submitted to arXiv.org⁴. The final publication is available at Springer via http://dx.doi.org/10.1007/s10009-017-0474-1

⁴ https://arxiv.org/
# Table of Contents

1 Introduction ................................................... 3
2 Related Work ................................................. 5
3 Overview of TestREx ......................................... 7
   3.1 Terminology .............................................. 8
   3.2 Typical workflow .......................................... 9
4 Implementation ................................................ 11
   4.1 Execution Engine ......................................... 11
   4.2 Applications .............................................. 12
   4.3 Images and Containers ................................... 12
   4.4 Configurations ............................................ 13
   4.5 Exploits .................................................. 15
   4.6 Report ................................................... 17
5 Evaluation .................................................... 18
6 Potential Industrial Applications ................................. 20
   6.1 Preparation and Training ................................... 20
   6.2 Design ................................................... 20
   6.3 Implementation and Verification ............................. 21
   6.4 Release and Response ...................................... 21
7 Conclusions ................................................... 22
   7.1 Lessons learned ........................................... 23
   7.2 Future work .............................................. 24
A Contributing to TestREx ...................................... 27
   A.1 Deploying an Application .................................. 27
   A.2 Creating Configuration Files and Building Containers ........ 27
   A.3 Creating and Running an Exploit ......................... 28
TESTREX: a Framework for Repeatable Exploits *

Stanislav Dashevskyi¹, Daniel Ricardo dos Santos¹,², Fabio Massacci¹, and Antonino Sabetta³

¹ University of Trento, Italy stanislav.dashevskyi@unitn.it, fabio.massacci@unitn.it
² Fondazione Bruno Kessler, Italy dossantos@fbk.eu
³ SAP Labs, France antonino.sabetta@sap.com

Abstract. Web applications are the target of many well known exploits and also a fertile ground for the discovery of security vulnerabilities. Yet, the success of an exploit depends both on the vulnerability in the application source code and the environment in which the application is deployed and run. As execution environments are complex (application servers, databases and other supporting applications), we need to have a reliable framework to test whether known exploits can be reproduced in different settings, better understand their effects, and facilitate the discovery of new vulnerabilities. In this paper, we present TESTREX – a framework that allows for highly automated, easily repeatable exploit testing in a variety of contexts, so that a security tester may quickly and efficiently perform large-scale experiments with vulnerability exploits. It supports packing and running applications with their environments, injecting exploits, monitoring their success, and generating security reports. We also provide a corpus of example applications, taken from related works or implemented by us.

Keywords: Software vulnerabilities, Exploits, Security testing, Experimentation

1 Introduction

Web applications are nowadays one of the preferred ways of providing services to users and customers. Modern application platforms provide a great deal of flexibility, including portability of applications between different types of execution environments, e.g., in order to meet specific cost, performance, and technical needs. However, they are known to suffer from potentially devastating vulnerabilities, such as flaws in the application design or code, which allow attackers to compromise data and functionality (see, e.g., [42,45]). Vulnerable web applications are a major target for hackers and cyber attackers [39], while vulnerabilities are hard to identify by traditional black-box approaches for security testing [13,28,43].

A key difficulty is that web applications are deployed and run in many different execution environments, consisting of operating systems, web servers,
database engines, and other sorts of supporting applications in the backend, as well as different configurations in the frontend \[28\]. Two illustrative examples are SQL injection exploits (success depends on the capabilities of the underlying database and the authorizations of the user who runs it \[12, Chapter 9\]), and Cross-site Scripting (XSS) exploits (success depends on the browser being used and its rules for executing or blocking JavaScript code \[45, Chapter 14\]). These different environments may transform failed attempts into successful exploits and vice versa.

Industrial approaches to black-box application security testing (e.g., IBM AppScan\[4\] or academic ones (e.g., Secubat \[24\] and BugBox \[32\]) require security researchers to write down a number of specific exploits that can demonstrate the (un)desired behavior. 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.

We need a way to automatically switch configurations and re-test exploits to check whether they work 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. \[2\] for testing exploit kits, but it is not available for testing web applications.

Our proposed solution, TestREX\[5\], 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 of applications to provide a real “playground” and an experimental setup where security testers and researchers can perform 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 exploit corpus is summarized in Table 1. Some of the exploits are taken from existing sources (e.g., BugBox \[32\] and WebGoat \[35\]), while others are developed by us. For the latter category, we focused on server-side JavaScript, because of its growing popularity in both open source and industrial usage (e.g., Node.js\[6\] and SAP HANA\[7\]) and, to the best of our knowledge, the lack of vulnerability benchmarks.

The rest of this paper is organized as follows. Section 2 describes and compares related work in the field of security experimentation; Section 3 presents an overview of TestREX; Section 4 discusses the implementation of the framework; Section 5 describes our evaluation of TestREX with testing various exploits, as well as using it as an educational tool; Section 6 lists potential uses of TestREX,

---

\[4\] http://www.ibm.com/software/products/en/appscan

\[5\] http://securitylab.disi.unitn.it/doku.php?id=testrex

\[6\] http://nodejs.org/

\[7\] https://help.sap.com/hana
2 Related Work

Security testing verifies and validates software requirements related to security properties such as confidentiality, integrity, and availability. Felderer et al. [18] argue that support for security testing is essential to increase its effectiveness and efficiency in practice.

Such support requires the development of experimental frameworks where developers can actually test and experiment with security bugs as advocated initially in [7,16,29]. Yet, this is far from trivial and only few papers in mainstream security conferences use experiments as their validation measure (e.g., at IEEE S&P 2015 only 3 papers out of 55 used experiments [8]).

Indeed, a number of issues should be tackled in order to correctly provide security experimentation setups. These issues include isolation of the experimental environment [2,5,6,32], repeatability of individual experiments [2,16], collection of experimental results, and justification of collected data [29].

The use of a structured testbed can help in achieving greater control over the execution environment, isolation among experiments, and reproducibility. Most proposals for security research testbeds focus on the network level (e.g., DETER [5], ViSe [2], and vGrounds [23]). A comparison of network-based experimental security testbeds can be found in [41]. On the application level there are significantly less experimental frameworks.

By using the taxonomy of Felderer et al. [18], the security testing process must be present in every phase of the lifecycle: model-based security testing between analysis and design; code-based testing and static analysis during development; penetration testing and dynamic analysis during deployment; and regression testing during maintenance. Application-based security experimental frameworks such as TestREx, BugBox, or WebGoat support security testers at the later stages of the lifecycle, namely deployment and maintenance.

Among the application level-frameworks, the BugBox framework [32] 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 concepts of execution isolation and environment flexibility. However, we needed to have more

Table 1: Available exploits in TestREx corpus

| Language                  | Exploits | Source      |
|---------------------------|----------|-------------|
| PHP                       | 83       | BugBox [32] |
| Java                      | 10       | WebGoat [35]|
variety in software configurations and process those configurations automatically. We have broadened the configurations scope by implementing software images for different kinds of web applications, and automatically deploy them.

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 [2]. 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 [29] 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. We summarize the discussed tools and approaches in Table 2.

Nowadays, many exploits are publicly available in websites such as Packet Storm [8] and Exploit-DB [9] or even integrated in penetration testing and exploitation frameworks such as Metasploit [10] and w3af [11]. There is also a growing black market for zero-day vulnerabilities and exploits [1] (these are vulnerabilities for which no patch is available yet). These exploits are usually provided “as is” and their reliability, i.e., their success rate against different targets, can vary widely. Exploit reliability has been studied in [15,21] and the conclusion of both studies is that most exploits have a low success rate when used “off-the-shelf”. Both studies were conducted for binary exploits (based on, e.g., buffer overflows). We are unaware of any work studying the reliability of web application exploits. Although we do not focus on increasing exploit reliability in this paper, TestREx can be used for testing it against web applications running with different configurations in different software environments.

Continuous system testing and quality control are also related to TestREx, since they can be used to automate regression testing and testing of different (evolving) versions of an application. Windmüller et al. [44] developed Active Continuous Quality Control (ACQC), an approach that uses automata learning to infer the behavior of web applications. Neubauer et al. [30] extended ACQC to support risk-based testing [19] by steering the ACQC process to increase risk coverage. TestREx differs from these works because it does not employ model inference, as the application and the test cases (exploits) are given by the user.

---

[8] https://packetstormsecurity.com/
[9] https://www.exploit-db.com/
[10] https://www.metasploit.com/
[11] http://w3af.org/
Table 2: Security testing and experimentation tools

The existing tools and approaches provide various functionalities with respect to deployment (e.g., from running on a local virtual machine to providing controlled environments on real hardware). Most security research testbeds focus on the network level, while on the application level there are significantly less experimental frameworks.

| Tool          | Description                                                                 | Exploit types                                                                 |
|---------------|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| BugBox [32]   | A corpus and exploit simulation environment for PHP web application vulnerabilities. | Selenium and Metasploit scripts in Python that exploit PHP application vulnerabilities. |
| MalwareLab [2] | A controlled environment for experimenting with malicious software.            | Programs that exploit various software vulnerabilities or malware kits.         |
| MINESTRONE [17] | A software vulnerability testing framework for C/C++ programs. The applications are deployed in virtualized environments via Linux Containers | Programs that exploit memory corruption, null pointer, number handling and resource leak vulnerabilities in C/C++ software. |
| DETER [5]     | A testbed facility that consists of a large set (around 400) of real machines. The resources infrastructure can be reconfigured on-the-fly upon request. | Programs that exploit various software vulnerabilities or malware kits.         |
| ViSe [3]      | A virtual testbed for reproducing and collecting the evidence of security attacks that is based on VMWare virtualization environment. | Multi-level attacks that include network tampering and software vulnerability exploitation. |
| SecuBat [24]  | Web vulnerability scanner that automatically scans live web sites for vulnerabilities using a web crawler infrastructure. | Specially crafted HTTP requests that exploit SQLi and XSS vulnerabilities.       |
| vGrounds [23] | A virtual playground for malware assessment, that is created on top of a physical infrastructure - a machine, a cluster or a multi-domain overlay infrastructure. | Malicious software such as virtual worms or malware kits.                       |

The integration of evolving model inference techniques into our framework is an interesting venue of future research.

3 Overview of TestREx

TestREx was designed to provide testers with a convenient environment for automated, large-scale experiments. We believe that TestREx is useful for developers as well. To support this claim, we give an example of a possible loophole in a bug fixing workflow of a hypothetical company:
A tester finds a bug and opens a new issue in a bug tracking system. She submits it as a test case described in natural language, explaining all pre-conditions and steps needed to reproduce the bug.

A manager assigns the issue to a developer. In order to pinpoint the source of the bug and understand how to fix it, the developer must reproduce the test case in his own setting. If the tester makes a mistake while creating the test case, the developer will be unable to trigger the bug. As a consequence, the developer rejects the fix request.

In the worst case, it might take a long time before the bug will be re-discovered and eventually fixed. In a better case, more resources are wasted if the tester has to re-describe the bug, and a manager has to re-assign the bug to a developer.

Using TestREx, the tester could create an “executable description” of a bug in the form of a script, and a packed execution environment that allows to instantly replay the situation that triggered the bug. Despite taking longer for the tester to initially describe the bug this way, it has many advantages over the natural language approach. First, the tester and the developer are ensured that the bug can be reproduced. Second, the test case can be kept as a template on which future tests can be developed, i.e., the first test is harder to describe, but future tests can reuse parts of the first one. Third, the test can be automatically added to a library of regression tests, to ensure that the same bug will be detected if reinserted in future versions of the application.

3.1 Terminology

Before we proceed, we introduce several general concepts that we use for further discussion:

- **Image** – a snapshot of an application configured to run in a certain software environment, including this software environment. An image can be instantiated into a container that a tester can interact with.

- **Configuration** – Configurations are used for creating images. We use this term as an intuitive meaning of a particular setup of an application and its supporting software components with particular values of setup parameters (configuration files, packages, etc.), as well as a set of instructions that are automatically executed in order to create an image in which these applications and components are “deployed”.

- **Container** – an instance of an image. This instance represents a certain state of an application and its software environment, that can be “run” for testing, and dismissed when testing is over. It can be either started using the pristine state of its base image (creating a new container, i.e., instance), or

12 Technically, these concepts are implemented using Docker (https://www.docker.io/) – we describe the implementation in Section 4. However, a different implementation may be obtained using traditional virtual machines to which these general concepts can be applied as well.
resumed from a certain existing state (re-using a container, that was already instantiated).

3.2 Typical workflow

An automated testbed should help security researchers in answering (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 the same environment?
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 the WordPress 3.2 application running with MySQL and, we would like to know whether (1) the same exploit works for WordPress 3.2 running with PostgreSQL; (2) the same exploit works for WordPress 3.3 running with MySQL; and (3) the same exploit works for WordPress 3.3 and PostgreSQL.

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

A key feature that we have sought to implement, is that the architecture of TestREx should be easily extensible to allow for the inclusion of new exploits, applications, and execution environments. Figure 1 shows a typical workflow when an application and the corresponding scripted exploits are deployed and run within TestREx:

1. A tester provides the necessary configuration for a specific image, including the application and software component files, and the scripted exploits to be executed (the latter is optional, as TestREx also supports manual testing).
2. The Execution Engine component of TestREx builds the image and instantiates the corresponding container.
3. The Execution Engine runs corresponding exploit(s) against the application container,
4. and monitors whether the exploit execution was successful.
5. After the exploit(s) are executed, the Execution Engine dismisses the corresponding container (optionally, further exploits may reuse the same container when the tester wishes to observe the cumulative effect of several exploits) and cleans up the environment.
6. The exploit(s) execution report is generated.

One of the main goals of TestREx is to make the testing process as automated as possible. Another important task is to make it possible to run applications and exploits in a clean and isolated environment. Therefore, we included the option of running every test against a clean state of the application. This gives the possibility to run tests in parallel (see the point 5 above).
The workflow of TestREx is straightforward: a tester provides configuration details of an application, its deployment environment, as well as the exploit scripts; TestREx automates the remaining actions, such as building and loading the environment, and running and monitoring the exploit.

Fig. 1: TestREx workflow

TestREx also includes some additional utilities. For instance, the Packing Module allows to package configurations in compressed archive files that can be easily deployed in another system running TestREx. Also, the Utilities module includes a collection of scripts to import applications and exploits from other sources, such as BugBox, and to manage images and containers.

Example 2. The inputs for Example 1 are instantiated as follows:

- **Application**: There are two applications of interest, each one is a set of `.html`, `.php` and `.js` files in a Wordpres folder.
- **Configuration**: There are four interesting configurations, 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.
- **Image**: There are two possible images, one with Ubuntu Linux distribution, Apache web server and MySQL database engine, and one with Ubuntu, Apache and PostgreSQL.
- **Exploit(s)**: There is only one exploit – a script that navigates to the vulnerable web page, interacts with it and injects a payload, simulating the actions of an attacker.
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 specific vulnerability in a given application.

When using the framework in a specific application, the exploit can be written by the tester or taken from a public source. In any case, the exploit code must be compliant with what we expect from an exploit, e.g., it must be a subclass of the BasicExploit class provided with TestREx, and contain metadata that specifies the target image and describes the exploit script (more details are in Section A.3 in the Appendix).

Aegis [12] extends the TestREx architecture to test run-time monitors, enforcing control-flow and data-flow integrity, as well as authorization policies in workflow-driven web applications. The synthesized monitors are deployed as Docker containers, and tests are implemented as Selenium scripts – as we have illustrated in Figure 13.1.

4 Implementation

TestREx is implemented in Python, mainly because it allows fast and easy prototyping and because of the availability of libraries and frameworks, such as docker-py to interface it with Docker (see below). Below we describe in details the implementation of each component of the framework.

4.1 Execution Engine

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

The single mode allows testers to specify and run a desired exploit against a container that corresponds to the chosen application image just once. This is useful when the 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 deployed in different software environments. A “.csv” report is generated at the end of the run.

To run applications and exploits in the batch mode, TestREx loops through a folder containing exploit files, and runs them against respective containers, generating a summary “.csv” report in the end. In this mode, the Execution Engine maps exploits to application images by scanning the metadata in each exploit, where appropriate target images are specified by the tester.

For manual testing, the Execution Engine instantiates a container based on the chosen application image, and returns the control to the tester (e.g., by

13 The steps of Aegis are: (1) start new containers with the appropriate applications; (2) run the workflows by using the Selenium script; (3) repeat the workflows with monitoring on; (4) capture results. At the end of the test session, the container can be destroyed and a new one re-created if workflows’ history are accounted for, and a pristine starting image is important for repeatability.
opening a web browser and navigating to the application, or returning a shell). No report is generated in this case.

The **Execution Engine** contains an additional setting for handing containers when chosen exploits are executed: it is possible to either destroy a particular container after the execution, in order to start with a “fresh” instance of the image for each exploit run; or to reuse the same container when its state has to be preserved, so that further exploits may have a cumulative effect that the tester wishes to observe.

### 4.2 Applications

Applications are packaged as “.zip” files containing all their necessary code and other 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 (they are shown in Table 5 and their corresponding vulnerability types are listed in Table 6, most of which are known real-world applications, only some of them being small artificial examples developed by us to explore security vulnerabilities typical for server-side JavaScript applications.

### 4.3 Images and Containers

Ideally, security testers should have the possibility of using various types of computing components and platforms, regardless of the type of underlying hardware and software that may be available.

To provide testers with the possibility of running applications in various environments in a flexible, scalable, and cost-effective manner, we employ software images (that are, implementation-wise, Docker images). Every such image represents a data storage for virtualized computing components or platforms, e.g., operating systems, application servers, database management systems, and other types of supporting applications.

Instead of creating virtual machines for applications and their software environments, we instantiate and run containers from corresponding images. These containers are based on the OCI standards, which are nowadays widely accepted in industry as a form of “lightweight virtualization” at the operating system level. They are sandboxed filesystems that reuse the same operating system kernel, but have no access to the actual operating system where they are deployed.

Some initial developments in this area were FreeBSD Jails, Solaris Zones, and Linux Containers. Currently, Docker is the *de facto* standard for con-

---

14 [https://www.opencontainers.org/](https://www.opencontainers.org/)
15 [https://www.freebsd.org/doc/handbook/jails.html](https://www.freebsd.org/doc/handbook/jails.html)
16 [https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm](https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm)
17 [https://linuxcontainers.org/](https://linuxcontainers.org/)
18 [https://www.docker.io/](https://www.docker.io/)
tainers. Docker provides a format for packing and running applications within lightweight file repositories that are called Docker containers. We use Docker to create images and instantiate containers.

Images are specified in Dockerfiles (a format defined by the Docker project) – these files represent configurations to which we refer in Section 3.1. Downloading generic software components and re-creating a Docker container from a corresponding image every time an application has to be run might be resource- and time-consuming. Therefore, we use image inheritance supported for Dockerfiles, creating several images for containers that hold generic software components, and can be reused by certain types of web applications. For instance, such images may encapsulate an operating system, a web server and a database engine, and their corresponding containers are instantiated only once. We provide some predefined images for common environments, using software components shown in Table 3. We use the following naming convention for such images: “<operating_system>-<webserver>-<database>-<others>”. In contrast, for images which actually contain an application to be tested (apart from generic software components) we use a different naming convention: “<application-name>__<software-image-name>”.

When the Execution Engine invokes an application image, the corresponding container will be instantiated and run using Docker. Then, depending on run settings (see Section 4.1), the container will be handled correspondingly when chosen exploits are executed (either destroyed, or reused for further exploit runs).

Table 3: Software components for generic images currently provided with TestREx

| Web server | DB engine | OS   |
|------------|-----------|------|
| Apache     | MySQL     | Ubuntu |
| Node.js    | MySQL     | Ubuntu |
| Node.js    | MongoDB   | Ubuntu |
| Tomcat     | MySQL     | Ubuntu |

Figure 2 gives an intuition on how an image for the WordPress 3.2 application can be composed with Dockerfiles: the image is created on the basis of two images combining Ubuntu OS with Apache web server and MySQL database.

4.4 Configurations

Implementation-wise, configurations correspond to the contents of Dockerfiles and supporting scripts that specify how an application can be installed and run in a container, including, e.g., prerequisites such as preloading certain data to a database, creating users, and starting a server. Additionally, configuration data for applications may include databases and application data.
Application images are composed of several “layers”: an operating system, a web server, and a database engine – the application itself is deployed on top. These components can be combined in all possible configurations supported by the application.

Fig. 2: Wordpress3.2__ubuntu-apache-mysql image

The configuration files must be placed in a separate folder under the configurations root folder (“<testbed_root>/data/targets/configurations”). We use the following naming convention to simplify matching configuration files with images that can be created using them: “<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 image that is intended for it.

Listings 2.1 and 2.2 present an example of a Dockerfile and a “run.sh” file, used to configure a WordPress 3.2 application within the “ubuntu-apache-mysql” image.

Listing 2.1: Dockerfile example

```bash
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
```

In Listing 2.1, line 1 specifies that the image for this application is built on top of the “ubuntu-apache-mysql” image. In lines 2 and 3, the application files
are copied to the “/var/www/wordpress” folder in the image and in lines 4 and 5, the “run.sh” script is invoked inside the container.

Listing 2.2: Shell script file example

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

In Listing 2.2, lines 2-5 are used to start the database server and pre-load the database with application data. Line 6 starts the Apache web server.

4.5 Exploits

Table 4 shows the classification of typical security flaws that might be present in both client- and server-side parts of a web application which can be tested with TestREx. The following security flaws may be present in web applications regardless of their implementation and deployment details. Yet, their successful exploitation strongly depends on the actual variant of deployment (e.g., MongoDB versus MySQL database, type and the version of the web server, etc.).

Table 4: Security flaws of web applications

| Security flaw                  | Description                                                                 | Technical impact |
|-------------------------------|-----------------------------------------------------------------------------|------------------|
| SQL/NoSQL injection           | User input is used to construct a database query and is not properly sanitized, allowing a malicious user to change the intended database query into an arbitrary one. Threats: Information Disclosure, Data Integrity, Elevation of Privileges. | Severe           |
| (SQLi/NoSQLi)                 |                                                                             |                  |
| Code injection                | Similar to SQLi/NoSQLi, however, instead of a database, user input is executed by a code/command interpreter. Malicious payload can be executed on both client and server, and may result into a complete takeover of the host machine on which the vulnerable application runs. Threats: Information Disclosure, Data Integrity, Elevation of Privileges, Host Takeover. | Severe           |
| Cross-site scripting (XSS)    | Each time a user-supplied data is being displayed in a web browser, there is a risk of XSS attacks: attacker can supply JavaScript code that either gets executed in a victim’s browser and stealing victim’s credentials or making actions on her behalf. Almost any source of data can be an attack vector (e.g., direct user input, data coming from a database, etc.). Threats: Information Disclosure, Elevation of Privileges. | Moderate         |

19 This classification is according to the OWASP TOP 10: [https://www.owasp.org/index.php/Top_10_2013-Top_10](https://www.owasp.org/index.php/Top_10_2013-Top_10)
| Security flaw                        | Description                                                                                                                                                                                                 | Technical impact          |
|------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------|
| Cross-site request forgery (CSRF)  | CSRF attacks take advantage of benign applications that allow attackers to act on their behalf: user is secretly redirected from a trusted page to attacker’s page, and user’s authentication information is used by an attacker. Applications that allow manipulations with DOM container of its pages are vulnerable. **Threats: Session/Credentials Hijacking.** | Moderate                  |
| Unvalidated URL redirects          | URL redirects instruct the web browser to navigate to a certain page. While this feature can be useful in many different contexts, developers should be careful and restrict user manipulations with a destination page: an attacker may conduct phishing attacks using a trustworthy website that has this vulnerability. **Threats: Open Redirect.** | Moderate                  |
| Sensitive data disclosure          | Sensitive/Personal data is attractive for attackers per definition, therefore the goal of most of attacks is to get a piece of such data. Since personal data is usually protected by law regulations, every such data flow in a web application must be protected against injection and interception attacks, as well as overly detailed error messages and application logic flaws that disclose context information to potential attackers. **Threats: Information Disclosure.** | Severe                    |
| Test code leftovers                | A tester may insert a piece of testing code into the application and forget to remove it upon release. This can lead to any kind of unexpected behavior: for example, anyone could get access to the application with a login 'Bob' and a password ‘123’ gaining full administrator access. Such forgotten pieces of test code are indistinguishable from maliciously crafted backdoors per se. **Threats: Backdoor.** | Severe                    |
| Using known vulnerable components  | If vulnerable versions of third-party components are used (e.g., an open source library) in a web application, an attacker can identify known vulnerabilities and perform a successful attack. In many cases, developers are not aware of all components they are using for their application. Vulnerable component dependencies aggravate the problem. **Threats: Potentially all of the above.** | May vary                  |

In TestREx, exploits may be any executable file that, when executed in a specified context, provide testers with unauthorized access to, or use of functionality or data within that context. Exploits include any sequence of steps that must be taken in order to cause unintended behavior through taking advantage of a vulnerability in an application and/or surrounding environment. For example, exploits may be used to provide access to sensitive data, such as financial
data or personal data. Exploits may hijack capabilities or other functionalities of applications and cause the applications to perform tasks that are not desired by authorized users, such as tracking user activities and reporting on these to the unauthorized user of the exploit. Other types of exploits may allow unauthorized users to impersonate authorized users.

Still, the above description of an exploit is quite vague, and may lead to having many automated exploit scripts that are not compatible due to various differences in their implementation (e.g., as a consequence it may be difficult to run them in a batch, and/or use them to produce a unified testing report). To avoid these potential problems, we implemented exploits as a hierarchy of Python classes that have the following minimal set of properties: (1) every exploit contains metadata describing its characteristics such as name, description, type, target application and container; (2) exploit classes must pass logging information and results of the run to the Execution Engine, providing a way for the Execution Engine to know that the exploit execution was successful.

We also incorporate the Selenium Web Driver\(^{20}\) for implementing exploits, as it can be used to simulate user/attacker actions in a web browser, and provides all necessary means to automate them. Additionally, it supports JavaScript execution and DOM interaction\(^{32}\). Every Selenium-based exploit in the framework is a subclass of the BasicExploit class, which encapsulates basic Selenium functionality to automate the web browser (e.g., “setUp()” and “tearDown()” routines, logging and reporting, etc.). 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. The success of an exploit run is also verified within the “runExploit()” method - this might be different for every exploit. This allows us to handle complex exploits that are not always deterministic, 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.

4.6 Report

Different context conditions may transform failed exploit attempts into successful ones, and vice versa. A given exploit test may include a number of possible combinations of applications, execution environments, and exploits, each of which may be configured in various ways. For example, an exploit that may be successful in exploiting a first application in a first environment may not be successful in exploiting that same application in a second environment, but may be successful in exploiting a second application in the second environment. Moreover, upon determining a success of a given exploit, it will be necessary to make some change to the application and/or execution environment, which will necessitate yet another testing (re-testing) of the previously successful exploit to ensure that the change will prevent future successful exploits.

\(^{20}\) [http://docs.seleniumhq.org/projects/webdriver/](http://docs.seleniumhq.org/projects/webdriver/)
Table 5: Applications in the corpus
The table shows the applications (real-world and artificial ones) that TestREx currently includes. The “Base Images” column specifies a base image which is used for creating a specific application image, and the “Source” column specifies the source from which we adapted exploits for these applications.

| Language | Applications | Base Images | Source |
|----------|--------------|-------------|--------|
| 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 | BugBox |
| Java     | WebGoat      | ubuntu-tomcat-java | WebGoat |
| Server-side JavaScript | CoreApp, JS-YAML, NoSQLInjection, ODataApp, SQLInjection, ST, WordPress3.2, XSSReflected, XSSStored | ubuntu-node, ubuntu-node-mongo, ubuntu-node-mysql | Our examples |

Therefore, we include a reporting functionality: whenever TestREx runs an exploit, it generates a report that contains the information about its execution. 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 the exploit and the target application names, identifier of an application-specific container, the type of the exploit, the exploit start-up status, the exploit execution result, and a comment field that may include other information that might be exploit-specific. Along with this report, the Execution Engine maintains a log file that contains information which can be used to debug exploits.

Notice that this reporting functionality is also considered critical by bad guys and is therefore present in almost all exploit kits available on black markets [25].

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.

Listing 2.3: An example of the report file entry after the exploit run

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

5 Evaluation

As a starting point in the evaluation of TestREx, we successfully integrated 10 example exploits from WebGoat [35], as well as the corresponding vulnerable
web applications. We also developed exploits for 7 specially crafted vulnerable applications, in order to demonstrate different types of exploits for SQL injection, NoSQL injection, Stored and Reflected XSS, Path Traversal and Code Injection vulnerabilities in applications that rely on server-side JavaScript: the examples for the latter two vulnerability types take advantage of vulnerabilities discovered in Node.js modules [33][34].

Table 6: Number of exploits in the corpus

*The table lists the number of exploits in the current corpus of TestREx, broken down by a vulnerability type and a programming language of the vulnerable portion of the source code that makes the exploitation possible.*

| Exploit          | #PHP | #Java | #Server JS |
|------------------|------|-------|------------|
| XSS              | 46   | 2     | 3          |
| SQLi             | 17   | 2     | 1          |
| Code injection   | 7    | -     | 1          |
| Auth. flaws      | 4    | 3     | -          |
| Info. disclosure | 2    | -     | -          |
| Local file incl. | 2    | -     | -          |
| CSRF             | 2    | -     | -          |
| DoS              | 1    | -     | -          |
| DB backdoor      | -    | 1     | -          |
| Param. tampering | -    | 2     | -          |
| Path traversal   | -    | -     | 1          |

As TestREx also supports the possibility of importing applications and exploits from other similar testbeds, we imported all exploits and corresponding applications from BugBox [32]. We used an automated script that copies the applications and exploits into the corresponding folders under TestREx, and creates identical configuration files for every imported application, using Apache as a web server and MySQL as a database server. We were able to run most of the BugBox [32] native exploits and collect statistics without modifying their source code. However, we had to create a specific base image (as specified in Table 5), as well as application images for BugBox applications.

Table 6 summarizes the types of exploits that we tested in various applications using TestREx: it shows that TestREx supports a variety of typical web application security flaws. This successful case study was instrumental for SAP to decide to put forward a patent application of the technology behind TestREx [37].

We also used TestREx for an edition of the Laboratory on Offensive Technologies course taught at the University of Trento, Italy. The goal was to teach MSc students enrolled into the Computer Science program about web application vulnerabilities and exploits by: (i) exploring the vulnerability corpus provided with TestREx, and (ii) enlarging the corpus with new exploits developed from
scratch by them, or ported from other sources. In the Appendix, we show an example of one exploit added to the corpus and detail the process of adding it. Adding an exploit consists of three steps: deploying an application, creating configuration files and building containers, and creating and running an exploit. We had the students focus on understanding the last step, the actual exploits, in the first moment, so that in a second moment they could write exploits for any application they wanted.

Although some students completed the assignments using TestREx, other students enrolled in the course preferred to develop system exploits (the two tracks were offered as possibilities at the beginning of the course). We were unable to draw relevant statistics about how the use of TestREx impacted the learning process, since that was the first edition of the course and there were only around 20 students in total (which is not statistically relevant, if we assume the usual 30 as the cut-off number). Nevertheless, we can affirm that TestREx helped in the preparation and teaching of the course as a source of ready examples that could be easily run and analyzed in students’ laptops (with varying configurations of hardware and software). In the future, we intend to analyze the use of TestREx with a larger number of students and possibly professional penetration testers, following an approach inspired by [9].

6 Potential Industrial Applications

There are several uses of TestREx that we are exploring in an industrial setting, covering different phases of the Secure Development Lifecycle [22] (SDL), and fulfilling the needs of different stakeholders. Below we summarize the activities in different phases of the SDL that can benefit from using TestREx. Part of this work has also been an object of a US Patent [37].

6.1 Preparation and Training

Part of a training toolkit. Security awareness campaigns, especially secure coding training, are commonly conducted in large enterprises, also in response to requirements from certification standards. From our own experience with TestREx, we believe that writing exploits may be an effective way to acquire hands-on knowledge of how security vulnerabilities work in practice and how to code defensively in order to prevent them. To quickly create a large corpus of artificially vulnerable applications for training purposes, it is possible to start from well-known applications and use vulnerability injection, as done in [20,36]. This way, we can easily create multiple examples for each category of vulnerabilities, with different levels of complexity for detection or exploitation.

6.2 Design

Security testing of cloud-based applications. One valuable use of TestREx is for cloud-based applications. In this scenario, a Cloud Service
Provider (CSP) provides the platform on which an Application Provider (AP) may run their applications. CSPs allow the same application to be provided on different platforms. However, such variations in context correspond to potential difficulties in ensuring reliable and complete security testing, because successful protection against an exploit in one context may prove unsuccessful in another context. In this setting, TestREx can provide highly adaptable, flexible, efficient, and reliable testing for different configurations, without requiring highly-specialized knowledge or abilities on the part of the security tester. For example, the security tester may be an employee of a CSP, which may wish to assure its customer APs that a secure platform is in place. In turn, the security tester may be part of the AP, who wishes to obtain independent testing of one or more platforms or platform providers.

6.3 Implementation and Verification

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 penetration testing of 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 penetration 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.

6.4 Release and Response

“Executable documentation” of vulnerability findings. When a vulnerability is found in a product, the ability 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 this practice may be erratic and complicate the security response. TestREx exploit scripts and configurations can be seen 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.

**Malware analysis.** Malicious third-party applications, also known as malware, are applications intentionally designed to harm their victims, by, e.g., stealing information or taking control of the victim’s computer. Malware in general, and especially web malware, are known to react differently to different environments (usually to avoid detection) [11,27]. Containers provide safe and repeatable environments for malware analysts to run their experiments. One possible use of TestREx is as a highly configurable sandboxing environment, where malware analysts can run potentially malicious applications in different configurations of an application to study its behavior. Another possible use is as a honeypot generator. Honeypots [10] are intentionally vulnerable applications deployed on a network to capture and study attacks.

**Security testing of third-parties components.** According to the Black Duck study [40] more than 65% of proprietary software vendors integrate Free and Open Source Software (FOSS) components into their applications. Since enterprise software typically has long maintenance and support lifecycles, older versions of FOSS components must be supported as well. When a new vulnerability in a FOSS component is discovered, the vendor has to verify whether it affects customers that are using different versions of the software applications which may also contain different (older) versions of the FOSS component. However, in this setting, traditional security testing with static and dynamic analysis tools and code reviews may be complicated [4,26,38]. In this scenario, TestREx can be used to test whether the customers whose versions of software applications are relying on older versions of a FOSS component are affected by a newly disclosed vulnerability in a much newer version of that component (which may not be the case, see [31]).

## 7 Conclusions

In this paper, we presented TestREx, a Framework for Repeatable Exploits that combines a way of packing applications and execution environments, automatic execution of scripted exploits, and automatic reporting, providing an experimental setup for security testers and researchers. TestREx provides means for the evaluation of exploits, as the exploits are reproduced in a number of different contexts, and facilitates understanding of effects of the exploits in each context, as well as discovery of potential new vulnerabilities.

We also provide a corpus of applications and exploits, either adapted from the existing works, or developed by us – we collected it to test the variety of applications and exploits that can be handled by TestREx.
7.1 Lessons learned

We can summarize the key lessons learned during the design and development of TestREx as follows: (1) build on top of existing approaches; (2) have a simple and modular architecture; (3) find reliable information on applications, exploits and execution environments in order to replicate them.

Building on top of the existing work, like we did with BugBox for the format of our exploits, and MalwareLab for the vulnerability experimentation design, was extremely valuable. This simplified our design and development time, and allowed us to quickly add a large corpus of applications and exploits on which we could test our implementation.

The functionality that TestREx offers can be achieved to a certain extent by separately using its individual tool components. For example, a tester can use a regular virtual or physical machine (or plain Docker) to deploy the software components of interest, and then either run a Selenium script or perform manual testing. However, she will have to become acquainted with all these tools and perform the experiments manually.

This approach only works if (i) the tester has only one single configuration to deploy and (ii) s/he has one single vulnerability to test, and (iii) the tester is both a security expert and a functional expert (i.e. knows the application and the actually deployed configurations). In all other cases, repeatedly using the individual tools would not scale.

Indeed, to create and test an exploit, a tester must first understand the “mechanics” of a vulnerability that can be exploited, or adapt existing exploits to specific conditions. Also, in many cases, publicly available exploit descriptions are vague, limited to a proof-of-concept (which may not necessarily work), and often lack information on how to reproduce them in a specific software environment.

Then this must be adapted to the actual configurations that are used in the company. This may be quite difficult due to the fact that the information on how to configure a certain application environment may not be detailed enough in the official documentation. Therefore, a functional expert may be more appropriate, but s/he may not have the security skills needed to deploy the vulnerability.

Instead, TestREx provides a completely automated solution in which the knowledge of different aspects can be “hidden away” from its different users (e.g., security testers might only need to know how to write and run exploit scripts, and everything else will be just “hooked up” and executed right out of the box; a functional tester would only need to write the configuration and just use the exploit scripts as provided).

Large scale usage of TestREx requires an initial investment which lies in creating a set of software configurations (images) that can be then reused company-wide: as we learned during the evaluation of TestREx, creating reliable software configurations is the most difficult part, while creating exploit scripts is comparatively easy. However, once this initial effort is invested, it becomes extremely easy to add new exploit scripts, run them in different combinations, combine different software images (we use Docker inheritance), and collect reports.
7.2 Future work

We intend to extend the architecture of TestREx to add support for plugins. Plugins (e.g., proxy tools, vulnerability scanners) could be used to facilitate activities such as pentesting, vulnerability analysis, and malware analysis, mentioned in section 6.

We would also like to expand the current vulnerability corpus by taking public exploits from, e.g., Exploit-DB and reconstructing the vulnerable environments in TestREx.

References

1. L. Allodi, M. Corradin, and F. Massacci. Then and now: On the maturity of the cybercrime markets the lesson that black-hat marketeers learned. IEEE Transactions on Emerging Topics in Computing, 4(1):35–46, 2016. pages 6
2. Luca Allodi, Vadim Kotov, and Fabio Massacci. Malwarelab: Experimentation with cybercrime attack tools. In Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET’13), 2013. pages 4, 5, 6, 7, 23
3. Andre Arnes, Paul Haas, Giovanni Vigna, and Richard A. Kemmerer. Digital forensic reconstruction and the virtual security testbed ViSe. In Proceedings of the 3rd International Conference of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA’06), 2006. pages 5, 7
4. Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities – does experience matter? In Proceedings of the 4th International Conference on Availability, Reliability and Security (ARES’09), 2009. pages 5
5. Terry Benzel. The science of cyber security experimentation: The DETER project. In Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC’11), 2011. pages 5, 7
6. Joan Calvet, Carlton Davis, Jose M Fernandez, Wadie Guizani, Matthieu Kaczmarek, Jean-Yves Marion, and Pierre-Luc St-Onge. Isolated virtualised clusters: testbeds for high-risk security experimentation and training. In Proceedings of 3rd USENIX Workshop on Cyber Security Experimentation and Test (CSET’10), 2010. pages 5
7. Thomas E. Carroll, David Manz, Thomas Edgar, and Frank L. Greitzer. Realizing scientific methods for cyber security. In Proceedings of the Workshop on Learning from Authoritative Security Experiment Results (LASER’12), 2012. pages 5
8. Jeffrey C. Carver, Morgan Burcham, Sedef Akinli Kocak, Ayse Bener, Michael Felderer, Matthias Gander, Jason King, Jouni Markkula, Markku Oivo, Clemens Sauerwein, and Laurie Williams. Establishing a baseline for measuring advancement in the science of security: An analysis of the 2015 IEEE Security & Privacy proceedings. In Proceedings of the Symposium and Bootcamp on the Science of Security (HotSos ‘16), 2016. pages 5
9. M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Fulcarin, and M. Torchiano. How professional hackers understand protected code while performing attack tasks. In Proceedings of the 25th International Conference on Program Comprehension (ICPC), 2017. pages 20
10. Kevin Zhijie Chen, Guofei Gu, Jianwei Zhuge, Jose Nazario, and Xinhui Han. Webpatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS’11)*, 2011. pages 22

11. Song Chengyu, Paul Royal, and Wenke Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security (HotSec’12)*, 2012. pages 22

12. Luca Compagna, Daniel R. dos Santos, Serena Elisa Ponta, and Silvio Ranise. Aegis: Automatic enforcement of security policies in workflow-driven web applications. In *Proceedings of the 7th ACM Conference on Data and Applications Security and Privacy (CODASPY’17)*, 2017. pages 11

13. Mark Curphey and Rudolph Arawo. Web application security assessment tools. *IEEE Security & Privacy*, 4(4):32–41, 2006. pages 3

14. Stanislav Dashevskyi, Daniel Ricardo Dos Santos, Fabio Massacci, and Antonino Sabetta. TestREx: a testbed for repeatable exploits. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET’14)*, 2014. pages 3

15. Maxwell Dono, Jonathan Risto, and Reginald Sawilla. Reliability of exploits and consequences for decision support. Technical report, Defence Research and Development Canada, 2015. pages 6

16. Eric Eide. Toward replayable research in networking and systems. *Position paper presented at Archive*, 2010. pages 5

17. Nathan S. Evans, Azzedine Benmeur, and Matthew Elder. Large-scale evaluation of a vulnerability analysis framework. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET’14)*, 2014. pages 7

18. Michael Felderer, Matthias Buchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one – security testing: A survey. *Advances in Computers*, 101:1 – 51, 2016. pages 5

19. Michael Felderer and Ina Schieferdecker. A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, 16(5):559–568, Oct 2014. pages 6

20. Jose Fonseca, Marco Vieira, and Henrique Madeira. Evaluation of web security mechanisms using vulnerability & attack injection. *IEEE Transactions on Dependable and Secure Computing*, 11(5):440–453, 2014. pages 20

21. Hannes Holm and Teodor Sommestad. So long, and thanks for only using readily available scripts. *Information and Computer Security*, 25(1), 2017. pages 6

22. Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006. pages 20

23. Xuxian Jiang, Dongyan Xu, Helen J. Wang, and Eugene H. Spafford. Virtual playgrounds for worm behavior investigation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID’05)*, 2006. pages 5, 7

24. Stefan Kals, Engin Kirda, Christopher Krügel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web (WWW’06)*, 2006. pages 4, 7

25. Vadim Kotov and Fabio Massacci. Anatomy of exploit kits. In *Proceedings of the Engineering Secure Software and Systems Conference (ESSoS’13)*, 2013. pages 18

26. Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, and Maurizio Morisio. Development with off-the-shelf components: 10 facts. *IEEE Software Journal*, 26(2):80, 2009. pages 22

27. Gen Lu. *Analysis of Evasion Techniques in Web-based Malware*. PhD thesis, University of Arizona, Tucson, AZ, USA, 2014. pages 22
28. Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172 – 1186, 2006. pages 3, 4

29. Roy A. Maxion and Kevin S. Killourhy. Should security researchers experiment more and draw more inferences? In *Proceedings of 4th USENIX Workshop on Cyber Security Experimentation and Test (CSET’11)*, 2011. pages 5, 6

30. Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. Risk-based testing via active continuous quality control. *International Journal on Software Tools for Technology Transfer*, 16(5):569–591, Oct 2014. pages 6

31. Viet Hung Nguyen, Stanislav Dashhevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering*, 21(6):2268–2297, 2015. pages 22

32. Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. Bugbox: A vulnerability corpus for PHP web applications. In *Proceedings of 6th USENIX Workshop on Cyber Security Experimentation and Test (CSET’13)*, 2013. pages 4, 5, 7, 17, 19, 23

33. Node Security Project. JS-YAML deserialization code execution. [https://nodesecurity.io/advisories/JS-YAML_Deserialization_Code_Execution](https://nodesecurity.io/advisories/JS-YAML_Deserialization_Code_Execution), 2013. pages 19

34. Node Security Project. st directory traversal. [https://nodesecurity.io/advisories/st_directory_traversal](https://nodesecurity.io/advisories/st_directory_traversal), 2014. pages 19

35. OWASP. Webgoat. [https://www.owasp.org/index.php/Category: OWASP_WebGoat_Project](https://www.owasp.org/index.php/Category: OWASP_WebGoat_Project), pages 4, 5, 18

36. Jannik Pewny and Thorsten Holz. Evilcoder: Automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC’16)*, 2016. pages 20

37. Antonino Sabetta, Luca Compagna, Serena Ponta, Stanislav Dashhevskyi, Daniel Ricardo Dos Santos, and Fabio Massacci. Multi-context exploit test management, 2015. US Patent App. 14/692,203. pages 19, 20

38. Izzet Sahin and Fatemeh Zahedi. Policy analysis for warranty, maintenance, and upgrade of software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(6):469–493, 2001. pages 22

39. Theodoor Scholte, Davide Balzarotti, and Engin Kirda. 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 (FC’12)*, 2012. pages 3

40. Black Duck Software. The tenth annual future of open source survey. [https://www.blackducksoftware.com/2016-future-of-open-source](https://www.blackducksoftware.com/2016-future-of-open-source), 2016. Last accessed 2017-07-11. pages 22

41. Evan Lawrence Stoner. A foundation for cyber experimentation. Master’s thesis, Florida Institute of Technology, 2015. pages 5

42. Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker’s Handbook: Discovering and Exploiting Security Flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007. pages 3, 4

43. Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA’14)*, 2014. pages 3

44. Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active continuous quality control. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE ’13)*, 2013. pages 6
A Contributing to TestREx

Here we describe in more detail the steps needed to add an experiment to TestREx, given an existing application. These steps consist of: adding an application; creating configuration files for images; instantiating containers; creating and running exploits. Again, we use WordPress 3.2 as the example application.

A.1 Deploying an 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”.

A.2 Creating Configuration Files and Building Containers

If there are no generic images that might be reused for creating a new image for the application set up, this image must be created in the first place. Configuration files for generic images are located under the “<testbed_root>/data/targets/containers” folder.

In our example, we create a generic image 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 2.4 and build it with the script located under “<testbed_root>/util/build-images.py”.

Listing 2.4: The Dockerfile for creating the ubuntu-apache-mysql generic image

```
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"]
```

As a next step, we create configuration files for the image that will hold the application, extending the above generic image. We create a new Dockerfile and a shell script file under the “<testbed_root>/data/targets/configurations/WordPress_3_2__ubuntu-apache-mysql” folder (see Listings 2.1 and 2.2 in the Section 4.4 for the code examples).

There is no need to manually invoke Docker for instantiating a container based on this image for running exploits or manual testing, as Execution Engine does it automatically.
A.3 Creating and Running an Exploit

Finally, we create an exploit for the Wordpress 3.2 application by creating a Python class under the “<testbed_root>/data/exploits” folder. As mentioned in the previous sections, to ensure integration with the Execution Engine, the new exploit class must be a subclass of the already existing BasicExploit class. As a last step, we specify the exploit’s metadata using the attributes dictionary, and specify the steps required to run the exploit within the “runExploit()” method (see Listing 2.5).

Listing 2.5: Wordpress_3_2_Exploit.py file contents

```python
from BasicExploit import BasicExploit
class Exploit(BasicExploit):
    attributes = {
        'Name': 'Wordpress_3_2_XSS',
        'Description': 'XSS attack in Wordpress 3.2',
        'Target': 'Wordpress3.2',
        'Container': 'ubuntu-apache-mysql',
        'Type': 'XSS'
    }
def runExploit(self):
    w = self.wrapper
    w.navigate("http://localhost:49160/wordpress/wp-admin/post-new.php?
                ?post_type=page")
    content_elt = w.find("content").clear()
    content_elt.keys("<script>alert("XSS!!")</script>")
    w.find("publish").click()
    w.navigate("http://localhost:49160/wordpress/?page_id=23")
    alert_text = w.catch_alert()
    self.assertIn("XSS", alert_text, "XSS")
```

Listing 2.5 shows the stored XSS exploit for the Wordpress 3.2 application. The script navigates to the login page of the Wordpress application, logs in as the administrator (the full list of steps is shortened in the listing for the sake of brevity), and creates a new post putting the `<script>alert(‘XSS’)</script>` string as the content. To verify whether the exploitation was successful, the script navigates to the newly created post and checks if an alert box with the “XSS” message is present.

In order to test whether the same exploit would work for different older or newer versions of the Wordpress application, a tester may reuse the configuration files of Wordpress 3.2. She only needs to provide the application files of that version. We indeed tested the exploit from the example on other versions of Wordpress: 4.2.15 and 4.8 (the latest available version). It works against the first, but does not work against the latter, as this vulnerability was fixed for the latest version.

Listing 2.6 shows the list of commands for different running modes in TestREX:

1. To run the application container for manual testing, a tester has to use the “–manual” flag and the corresponding application image. TestREX will run the container and halt, waiting for the interrupt signal from the tester. In
In this mode, when the container is up, the application can be accessed from a web browser by navigating to [http://localhost:49160].

2. In the single mode a tester can select a specific exploit and run it against a specific application image.

3. In the batch mode for a single application, a tester has to specify the running mode as “--batch”, and select the desired application image. TestREx will invoke a Docker container for the image, search for the exploits that are assigned to the application (through exploits’ metadata), and run all of them one by one.

4. Finally, if a tester specifies nothing but the “--batch” running mode, TestREx will invoke containers for all application images that are currently in the corpus, and run all corresponding exploits against them.

Listing 2.6: Running modes in TestREx

#1: Manual mode
sudo python run.py --manual --image
    [app-name]__[image-name]

#2: Single exploit mode
sudo python run.py --exploit [exploit-name].py
    --image [app-name]__[image-name]

#3: Batch mode for a single application
sudo python run.py --batch
    --image [app-name]__[image-name]

#4: Batch mode for all applications
sudo python run.py --batch

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