
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INFORMATION SYSTEMS**
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**


# WEB APPLICATION PENETRATION TESTING AUTOMATION

**AUTOMATIZACE PENETRAČNÍHO TESTOVÁNÍ WEBOVÝCH APLIKACÍ**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** — **Bc. DANIEL DUŠEK**
**AUTOR PRÁCE**

**SUPERVISOR** — **Ing. JAN PLUSKAL**
**VEDOUCÍ PRÁCE**



**BRNO 2019**

**Brno University of Technology**
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

# Master's Thesis Specification

21678

Student: **Dušek Daniel, Bc.**
Programme: Information Technology Field of study: Information Systems
Title: **Web Application Penetration Testing Automation**
Category: Networking
Assignment:

1. Study approaches, methods and existing software used for Web Application penetration testing. Focus on techniques and ways of penetration testing that can be automated. Select a suitable testing set of Web Applications upon which you will be able to demonstrate studied approaches.
2. Design suitable, generally applicable approach to Web Application penetration testing focused on retrieving information about target Web Applications that can be extracted by non-permanent and non-destructive interactions.
3. Design and implement an automated tool that follows the approach proposed in (2) for Web Application penetration testing.
4. Evaluate the implemented tool against the testing set selected in (1).
5. Create user documentation for the implemented tool, do not omit examples of proper usage. Outline possible ways of future extensions and discuss their benefits.

Recommended literature:

1. Zalewski, M. (2012). The tangled Web: A guide to securing modern web applications. No Starch Press.
2. Weidman, G. (2014). Penetration testing: a hands-on introduction to hacking. No Starch Press.
3. Stuttard, D., & Pinto, M. (2011). The web application hacker's handbook: Finding and exploiting security flaws. John Wiley & Sons.

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Pluskal Jan, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: October 30, 2018

# Abstract

This work has two goals — to propose a generally applicable approach to web application penetration testing that is non-destructive to a target application, and to implement a tool that will follow it. The proposed approach has three phases. In the first phase, a tester gathers and adheres to the testing requirements (including the non-destructiveness), prepares a tool set and starts the reconnaissance. In the second phase, additional testing tools are used to process collected information and to verify vulnerabilities and provide conclusions. During the third phase, a final report is generated. The implemented tool is built as a collection of modules that are capable of the detection of reflected XSS, hidden query string parameters, resource enumeration and server misconfigurations detection. In comparison to Acunetix vulnerability scanner, the implemented tool performs just as well in the reflected XSS detection and outperforms the Acunetix in hidden resources enumeration. This work also brings a proof of concept implementation of a tool for Pastebin.com side-channel monitoring.



# Abstrakt

Tato práce má dva cíle — navrhnout obecně aplikovatelný přístup k penetračnímu testování webových aplikací, který bude využívat pouze nedestruktivních interakcí, a dále pak implementovat nástroj, který se tímto postupem bude řídit. Navrhovaný přístup má tři fáze — v první fázi tester posbírá požadavky pro testovací sezení (včetně požadavků na nedestruktivnost) a připraví si nástroje a postupy, kterých při testování využije, následně začne s průzkumem. V druhé fázi využije dodatečných nástrojů pro zpracování informací z předchozí fáze a pro ověření a odhalení zranitelností. Ve třetí fázi jsou všechny informace překovány ve zprávu o penetračním testování. Implementovaný nástroj je postavený na modulech, které jsou schopny odhalení reflektovaného XSS, serverových miskonfigurací, skrytých adresních parametrů a skrytých zajímavých souborů. V porovnání s komerčním nástrojem Acunetix je implementovaný nástroj srovnatelný v detekci reflektovaného XSS a lepší v detekci skrytých zajímavých souborů. Práce také originálně představuje nástroj pro sledování postranního kanálu Pastebin.com s cílem detekce utíkajících informací.



# Keywords

Penetration testing, web applications, automation, information security, security, open source inteligence, automated security scanning.

# Klíčová slova

Penetrační testování, webové aplikace, automatizace, informační bezpečnost, veřejně dostupné informace, automatizované testování počítačové bezpečnosti.


# Reference


DUŠEK, Daniel. *Web Application Penetration Testing Automation*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Pluskal

# Web Application Penetration Testing Automation

## Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Jan Pluskal. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Daniel Dušek
May 13, 2019

## Acknowledgements

I would like to thank my thesis advisor and supervisor Ing. Jan Pluskal. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing.

# Contents

# Chapter 1

# Introduction

One of the today's most interesting areas for penetration testing is the realm of web applications. A great part of the day to day tasks that most people face can and is done either through the web or other service that utilizes the web platform indirectly. With this relatively new trend the need for securing and security testing these platforms arises.

This work aims to study existing approaches, techniques and software used for web application penetration testing and then propose new theoretical approach to it. The new approach hopes to be both generally applicable and possible to automate, while utilizing *non-permanent* and *non-destructive* actions and operations. Using this type of actions is motivated by the idea of later automation and their automated execution without the risk of damaging the target application while doing so.

Proposed approach aspires to enable automation, which will decrease the amount of manual work that would otherwise needed to be performed by a tester. This saves their time and frees their hands to focus on other tasks with higher priority, or the tasks that would be normally out of the tested scope. As the approach assumes that the automation will take place, it also considers enabling a tester to observe, follow, and process higher volume of data through automated means, which then also increases the scope covered by testing.

Subsequently, the tool for web application penetration testing that follows the proposed approach is designed, implemented and tested with a vision of easing the work of a penetration tester. The expected output of the tool is not purely the report of discovered vulnerabilities, but also recommendations for a tester relevant to a tested application.

Chapter 2 of this thesis summarizes the current state of the web application penetration testing. This includes typical actors that conduct it, existing approaches and methods such as *Zero Entry Hacking* [8] or *OWASP Web Application Security Testing* [29], common techniques and existing software used for the penetration testing. The chapter closes up with a specification of testing set of web applications against which the implemented tool is to be tested later on in the thesis.

Chapter 3 is dedicated to designing the suitable and generally applicable approach to web application penetration testing, taking advantage of the already mentioned *non-permanent* and *non-destructive* actions and operations.

In chapter 5 the final product of this thesis, *ReconJay* tool is designed. Later in the chapter its implementation is presented and the important parts are highlighted. The tool is

comprised of modules that support discovery of a reflected XSS vulnerability, hidden resources and parameters enumeration and IIS/VCS misconfiguration detection. Apart from that, a standalone module named *PastebinTracker* for monitoring the *Pastebin.com* side-channel is designed, implemented and presented. Last part of the chapter 5 presents two hypothetical case studies that explains possible use case of the both implemented tools.

Chapter 6 evaluates both implemented tools from multiple perspectives. Evaluation phase begins with a general functionality testing of the implemented tools and then continues with evaluation of each of the implemented modules. After that a comparison to a commercial *Acunetix* vulnerability scanner is presented. To wrap up the evaluation phase, *ReconJay* tool is run against testing sets of web applications specified in chapter 2.

# Chapter 2

# Web Application Penetration Testing Today

This chapter aims to cover the current state of web application penetration testing in relevance to the overall goal of the thesis.

Reading this chapter will introduce the reader into some of the causes of vulnerabilities in modern web applications and the reasons for their prevalence. The reader will also learn about the two approaches (methods) of penetration testing commonly used in the field of web application security. This chapter also features frequently used web application penetration testing techniques and discusses their properties and possible automation. Ending of this chapter presents a set of target web applications used for testing the utility implemented in practical part of the thesis. Selection process of testing web application set is also mentioned.

First part of this chapter presents current state of web applications that are all typically published on the today's Internet.

Second part discusses typical actors operating in the realm of *Web Application Penetration Testing*, their possible motives and agendas, as well as tools and approaches deployed.

Third part of this chapter takes a closer look at the two most common and most used penetration testing approaches (methods).

Fourth section focuses on common web application penetration testing techniques while paying special attention to those techniques that can be automated. Strengths, weaknesses and potential to be performed without destructive consequences are discussed.

Fifth section of this chapter studies existing software used for penetration testing of web applications. When possible, chaining of such software to achieve greater automation is proposed. Strengths and weaknesses are discussed and possibility of future extensions and improvements is mentioned.

Sixth and final section of this chapter presents a set of target web applications against which the utility implemented in practical part of the thesis is tested, including the selection process of applications put into this set.

## 2.1 Web Applications Today

In the early days of the Internet, websites and web applications were mainly static HTML documents created by first Internet users, to be shared with others. A lot has changed since then and the Internet has become more about communication and user cooperation, than about passive document sharing and reading. This shift towards more active user engagement and contribution is also known as a transition from *Web 1.0* towards *Web 2.0* [7].

The *Web 2.0* drastically changes the way web applications are built and used. The focus is now more on bringing people together (social networks like Facebook, Twitter, Instagram, and others) and on allowing them to communicate efficiently (web email clients, team communication tools like MS Teams or Slack, or, of course, integrated chat into aforementioned social networks) [7]. Online and mobile banking applications are also becoming a norm [1].

In order to power the *Web 2.0*, loads of new languages and technology means were invented. Use of Javascript on modern web is way more common then it used to be, new "hot" frameworks and libraries are released almost every day and web developers are constantly on the lookout for the new technology or the trick that would enable them to serve content to their users just a little bit faster.

Modern ways of developing web application — like for example using the Node.js — come out of the box with an available package manager that enables developers to use other developer's code in their applications very easily. Finally, the produced web applications are way more complex and comprise of way more source code than they used to.

Today's web applications are also way more connected to each other. One of the very common patterns that can be spotted "in the wild" is web application referencing sources from different applications located on the internet. A great example of such behavior is the *Google Analytics Javascript* code used for tracking user activity on the web. Other popular 3rd party Javascript source files frequently included in websites are various kinds of support chat for typical e-commerce sites, or tools enabling additional functionality for visually impaired users.

A lot of applications that used to reside within a user's computer, as desktop applications, is also slowly replaced by their web-based alternatives or clones. Very frequently out of convenience — if the user owns multiple devices, it is easier to use cloud-based web application instead of installing the application on each and every device. Though it may not be too obvious, this replacement has one important side effect — a lot of data that would normally stay off the web (and all the third-parties) are now transmitted and stored to new places for which the data may have not been intended.

While all this heavily improves the user experience for both the developers and the standard users, with all the new technology being introduced, the attack surface on web applications grows larger every day. And so is the dependence of newly created web applications on frameworks and libraries.

All of the mentioned above substantially increases the attack surface of a typical modern web application and should be paid special attention in the development phase. Combined with the everlasting competition between browser vendors for new HTML5, CSS3, and Javascript features [40], which also led to vulnerabilities in the past [40], web application security becomes a very broad and current topic.

## 2.2 Web Application Penetration Testing Today

At the time of writing, most of the internet facing web applications is likely impacted by some form of penetration testing on a daily basis — be it in the development phase from well-intended reasons by members of the development team, in the production by curious security-savvy visitor, or by automated security scanning crawlers written for various reasons. The need to secure web applications against the misuse by the users and potentially by the hackers is generally acknowledged.

Actors conducting penetration testing of web applications are often classified into three categories [3, 36] (see below). The author of this thesis, based on his experience, proposes one additional category (the last bullet point):

- Researchers,
- malicious hackers,
- script kiddies,
- automated/autonomous systems.

Each of the mentioned categories has its representative members with various motives for their actions. Prototypes of actors from these categories are briefly introduced and discussed below. Note that this introduction does not aim to cover every possible actor and their motivation, but rather to showcase the most commonly present actors and offer the author's thoughts on their possible motivation and reasoning behind it.

### Actor: Researcher

Web application researchers, sometimes also called "white-hats", are interested in ethical hacking and penetration testing with a goal of improving the security state of the system under test, while staying within the law [30]. Very often they study already existing research and techniques and then try to build on top of them, to take it a bit further [19].

Some of them may be motivated by the security bug bounty put out by software companies. Others may be motivated by improving their own knowledge, information security field in general or the combination of already mentioned. Very often, researchers publish their research in dedicated security conferences around the world.

Researchers very often write their own software tools that help them conduct penetration testing. These tools are sometimes open-sourced and shared among other researchers so they can be expanded and improved by the community. Example of such tools could be ZAP, WebScarab or CAL9000[1] (all developed as a part of the OWASP initiative). There are also commercial and community-free tools used by many, that are designed to ease penetration testing and research, such as BurpSuite[2].

[1] Available at https://www.owasp.org/index.php/Appendix_A:_Testing_Tools
[2] Vendor's website: https://portswigger.net/burp

### Actor: Malicious Hacker

Malicious hackers, often called "black-hats", are actors that conduct penetration testing with malicious intentions, very frequently motivated by financial or other personal gains [32]. They employ various creative ways necessary to successfully penetrate their target while paying extra attention to covering their tracks and not being discovered.

These actors may come from very different backgrounds such as criminal organizations, various government intelligence agencies, terrorist and hacktivist groups, but they can also be working alone [18]. Motivation highly depends on their background — for criminal organization hackers it could be money or access to sensitive information that could be used for extortion or avoidance of capture, for government intelligence agency hackers it could be sensitive information and surveillance.

Tools being used are again dependent on the organization or background from which the hackers come. Same as *white hat* hackers, *malicious hackers* certainly write a lot of their own utilities and tools to simplify hacking, and they just as well use commercial and community-powered tools too.

### Actor: Script Kiddie

Actors falling within this category are mostly people with insufficient computer skills relying on other people's work that they can use to exploit and penetrate the systems. They tend to use software developed by more qualified and more advanced hackers to attack their targets. Due to their lack of skills and knowledge in the field, they often do not take the necessary precaution to avoid getting caught [31].

They are motivated by getting attention for their "hacks" and defacements of web applications and similar, many times by making a name for themselves [23].

Generally, any tool that provides clear enough user interface and advertises that enables hacking can be in script kiddy's repertoire. There are recorded cases when tools developed likely by black-hats, targeted at script kiddies, were introducing malware into script kiddies' computers and networks [14].

### Actor: Automated/Autonomous System

The last category of *penetration testing actors* mentioned in this thesis is the category of automated or event autonomous systems. Such systems crawl the internet facing web applications and scan them for vulnerabilities. These systems can be deployed by any of the previously named categories of actors to follow their agendas.

Such systems can vary in their complexity and nature. Commercial general purpose vulnerability scanners such as Netsparker[3] or BurpSuite scanner are representatives of the *automated* group. Penetration testers deploy these tools against their target instance and launch the scan. Once the scan finishes, they are provided with an overview of discovered vulnerabilities and places where they were discovered. The results of these scanners cannot be blindly trusted [26] and it is up to penetration testers to reproduce and verify them.

[3]Vendor's website: https://www.netsparker.com/

Certain systems deployed to the Internet are not only automated but also autonomous. Example of such system is the *Andromeda/Gamarue* botnet that targeted *Wordpress* instances and tried to break into their administrations [37]. Once it broke in and gained control, it used the instance to send out spam [6].

Other systems of similar nature exist, with various targets and agendas, scanning and exploiting vulnerabilities in well-known web content management applications [35], and very likely also systems that crawl the Internet and look for the well-known vulnerability classes.

Author of this thesis believes that automated and autonomous actors will be much more prevalent in the future as the number of internet-connected devices grows bigger every day. With the growing number of Internet-connected devices grows also the number of devices that can be infected and abused for malicious purposes, such as running vulnerability scanning bots mentioned above.

## 2.3 Penetration Testing Approaches and Methods

Following text introduces two existing penetration testing approaches (also referred to as testing methods). The first one being generally applicable to any form of penetration testing, while the other one being developed specifically for web application penetration testing.

### Zero Entry Hacking (5 phases)

The most commonly used and cited approach to penetration testing states 5 significant steps, or phases, in which the penetration testing is executed. This approach sometimes called *Zero Entry Hacking Methodology (ZEH)* [8] focuses on getting from broader phases to the more specific ones. Opinions on the actual number of phases covered in *ZEH* differ, as certain sources suggest only 4 steps, omitting the last one. Reasons for omitting the fifth phase is briefly introduced below.

The four plus one steps, as described by Patrick Engebretson in the book *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy* [8], are the following:

- **Reconnaissance** (also "Information Gathering", "Recon") — Penetration tester collects available information about their target. The goal of this phase is to collect as much relevant information about the target as possible.
- **Scanning** — Manual and automated tools are used to scan the target for vulnerabilities and weaknesses in its defense.
- **Exploiting** — Penetration tester attacks discovered weak spots and tries to exploit vulnerabilities. The usual goal is to gain access either to administrative-level privileges on the target or to the sensitive data stored there.

- **Maintaining Access** — Once the target is breached, penetration tester establishes the way to access the target again in the future. This may often include installing backdoors or creating new administrative accounts on the target machine.
- *Covering Tracks (also "Hiding" or "Destroying evidence")* — After successful penetration and ensuring there is a way to maintain access in the future, penetration testers may take further action to cover the fact that penetration occurred. This may, for example, include deleting or altering logs from the time window when phases 1 to 4 took place. Some sources do not recognize or state this phase, from ethical and educational reasons.

To illustrate the decreasing scope and the decreasing amount of information gathered across the phases of *ZEH*, the inverted triangle diagram is sometimes used [8].

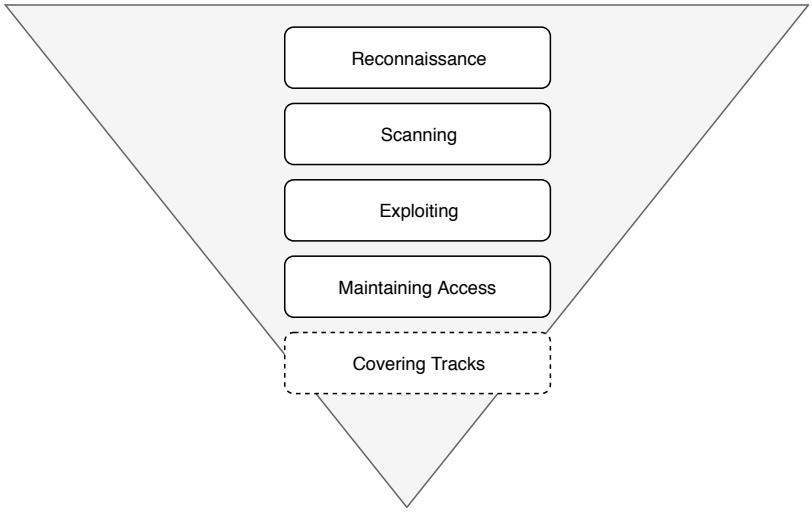


Figure 2.1: The Zero Entry Hacking Penetration Testing methodology visualized as a reverse triangle diagram, illustrating the decreasing scope and amount of information when progressing from the initial phases to the final phases [8]. The diagram is extended by the fifth phase dedicated to covering the tracks of successful penetration.

### OWASP Web Application Security Testing

OWASP (*Open Web Application Security Project*) is an open community dedicated to helping companies to conceive, develop, acquire, operate, and maintain web applications that can be trusted [29]. To help with these efforts, the *OWASP Testing Guide* (OTG) was created and later several times revised, with the latest version 4.0 from 2014 [27].

The OTG is comprised of 3 significant chapters covering a great range of topics from the web application security. The first chapter of the guide, *The OWASP Testing Framework*, targets mainly activities relevant to application development, maintenance, and operation, whereas the second chapter, *The Web Application Security Testing*, presents OWASP methodology for web application penetration testing. All of that is then closed by chapter 3, *Reporting*, which advises the tester on how to report their discoveries. Further down in this section of the thesis, methodology from chapter 2 of OTG is briefly introduced.

**Phases of penetration testing by OTG v4.0** [27]:

- Passive mode phase
- Active mode phase:
  - Information Gathering
  - Configuration and Deployment Management Testing
  - Identity Management Testing
  - Authentication Testing
  - Authorization Testing
  - Session Management Testing
  - Input Validation Testing
  - Error Handling
  - Cryptography
  - Business Logic Testing
  - Client Side Testing

Unlike the previously mentioned methodology, the OTG proposes a different approach to testing for and discovering security issues. It divides penetration testing into two significant phases — passive mode and active mode phase.

In the passive mode phase, penetration tester is supposed to interact and "play around" with the system under test, while passively collecting information about the possible ways to communicate with the application. Special attention should be paid to so-called *gates or access points* of the system. Use of tools to collect useful information about the target application in the background, as tester uses the application, is recommended.

Then, in an active mode phase, the tester is to start testing for 11 OWASP defined subsections and their respective methodologies. Each of the subsections of active mode phase has its methodology defined. These methodologies are usually fairly concrete about the vulnerabilities and ways how to test given subsection, very often even recommended input is suggested.

## 2.4 Common Penetration Testing Techniques

Number of interesting penetration testing techniques is presented in this section. Special attention is paid to techniques that can be automated and to such techniques that are non-destructive and make no permanent changes to the target web application.

### Static Code Analysis

Static source code analysis is a process of inspecting source code without executing it, usually performed by an automated tool [39]. There are different reasons for performing the static source code analysis, such as *type checking*, *style checking*, *program understanding*,

*program verification*, *bug finding* or *security reviews* [5]. No matter the reason, the tool performing static analysis will always check for pre-defined patterns contained within the source code and then report (with varying precision) its observations relevant to its reasons [9].

When web browser is used to visit website, prior to rendering the website's user interface, its source code needs to be requested from server that hosts the website [12]. Most commonly retrieved source code formats today include, but are not limited to, *HTML* files, *XML* files, *CSS* files and *JavaScript* files.

The last mentioned, *JavaScript* files, can be interesting from penetration testing perspective, as modern single page web applications take advantage of *JavaScript* to communicate with the backend application, rather than using the traditional request-response process [16]. It is possible, for penetration tester to learn about backend application's endpoints by performing static analysis of *JavaScript* source files and looking for all calls to functions that open connection to remote server. In serious cases of flawed implementation, analysis of the source code can also reveal other useful information, for example, like application secrets or vulnerable coding patterns.

Static code analysis technique is a great candidate to be used as a way of penetration testing that is non-destructive and non-permanent. Retrieval of *JavaScript* source files must be executed by using HTTP verb `GET`, which does not affect requested object [10]. Finally, static analysis was from the very beginning meant to be performed by machine and not by human [39], which once again, makes it a perfect candidate for use in automated penetration testing.

### Subdomain Enumeration

Sometimes it can be useful for penetration tester to discover existing subdomains of the target application to gain additional information, that could be later helpful in finally breaching the target. Existing subdomains does not necessarily have to be explicitly mentioned anywhere in the production application, because services hosted there have no connection to application with which the regular user is supposed to interact. Such subdomains, for example, could be used as a testing environment for development team, or maybe even for different internal purposes. Sudbdomains of this kind are of particular interest to the penetration testers.

Multiple approaches to subdomain enumeration exist, the most well-known are the following [15]:

- *Zone Transfer Request to DNS* — It is possible to try issuing `AXFR` request to DNS server to obtain the contents of the whole zone. In real life, this approach is usually not applicable, as it is recommended best practice to deny *zone transfer* requests from unauthorized sources [21].
- *Searching the Google Index* — Searching for `site:domain.com` will reveal all pages indexed by *Google* on given domain, including subdomains[13]. This process can not be automated, because google forbids and prevents any automated use of its searching services in their *Terms of Service*[4].

[4] Available online: https://policies.google.com/terms

- *Certificate Search*[5] — If subdomain or subdomains are using HTTPS, they can be discovered by searching the *Certificate Search* for their parent domain.
- *Brute Forcing/Dictionary Attack* — Trying various well-known subdomain names, dictionaries of words or straight up brute forcing subdomain names is also possible, but time consuming way to discover existing subdomains. Use of a good subdomain name dictionary can theoreticaly improve the performance.

One, most notable tool, that combines and utilizes the mentioned and the few other approaches is called *Sublist3r*[6]. The *Sublist3r* automates the process completely — penetration tester only needs to run the tool on the target application.

Again, subdomain enumeration uses only HTTP `GET` verbs to make enumeration requests, which does not affect requested object [10].

### Subdomain Takeover

There are cloud service providers that offer application hosting services with possibility of serving the application content on the customer's own canonical domains or subdomains.

This is the best explained with an example: An imaginary cloud service provider *Hosting Company* provides the application hosting and deploys these applications by default on URL *app-name.hosting-company.com*, but also allows the customers to point these URLs to their own canonical domains, such as for example *app-name.customer.com*.

To point the hosted application to the customer's canonical domain, DNS `CNAME` record can be used [25]. Customer typically sets the `CNAME` record for their canonical domain *app-name.customer.com* to the subdomain hosted by *Hosting Company*. Whenever the real visitor then inputs the *app-name.customer.com* address into their browser and hits the enter, content served from *app-name.hosting-company.com* is shown.

Subdomain takeover attack relies on the possibility that sometime in the future, the domain of the application (owned by the cloud service provider) will become available for registration again. That may happen when the customer decides that the application is no longer needed and terminates the contract with cloud service provider. When they do this and forget to remove the `CNAME` record set up for their *app-name.customer.com* domain, they become vulnerable. Attacker can register the expired domain with cloud service provider and start serving their content on the victim's canonical domain, because the `CNAME` record still exists.

Practical example of attack that could be carried out by the attacker is for example displaying a logon form on the vulnerable domain and phishing user credentials. The affected users have no way of knowing that they are not really entering their credentials into the fake logon form.

Examples of the real cloud service providers that offer this functionality at the time of writing this thesis are: *Heroku*[7], *GitHub*[8] or *Amazon Web Services*[9].

[5] Available online: `https://crt.sh/`
[6] Tool is available online: `https://github.com/aboul3la/Sublist3r`
[7] See `https://www.heroku.com/`
[8] See `https://github.com/`
[9] See `https://aws.amazon.com/`

Once again, testing for subdomain takeover vulnerability is possible without issuing destructive or permanent trace leaving interactions and the testing can be automated.

### Locating Hidden Resources

For the purposes of this technique, any file available on the target application is considered to be a *resource*. A *hidden resource* is a file to which there is no direct link written in the source code that is presented to a visitor.

In some situations, these files may get into the production environment by accident — for example, the developer copies whole folder with application source files to the production FTP folder, including all the "invisible" files and directories — as well as by negligence or by the lack of knowledge — a developer does not expect anyone to specifically try to visit addresses to which there are no hyperlinks from the website.

Real life examples of such files include:

- *Readable configuration files* — May contain passwords, secrets, access tokens or other sensitive information.
- *Version control files and folder* — Can leak the actual application code to the attacker, including the incremental history of the application source. Examples of these files and folders may be a `.git` or `.gh` folder, files with .svn extension, and similar.
- *Server logs* — If server or application logs are stored in the predictable location (or the location address can be discovered, e.g. through enabled directory browse), information about server's or application's inner workings can be revealed.
- *IDE configuration files* — Another valuable source of information about the target web application are the IDE configuration files. For example when inspecting the *JetBrains*[10] IDE configuration file located at `.idea/workspace.xml`, extraction of the complete web application directory and file structure is possible.

There is also a special kind of files that can be, by the definition mentioned at the beginning of this section, considered a hidden resource — files like `robots.txt` or files located under `.well-known` sub-directory. These files contain instructions meant to be processed by machines, for example what pages should and should not be indexed, or what are the restrictions on ways of accessing certain locations [28, 20]. It is possible for a tester to extract valuable information from them too, if the files' purpose was misunderstood by the developers.

Discovering these files is a trivial task of sending HTTP `GET` requests and checking the response codes. When the response code is the `200 OK`, the contents of the file are returned and can be reviewed by a tester, otherwise the file is either not available (`404 Not Found`) or the access is not possible (various reasons and codes). Penetration tester can specify their own files and directories of interest, or they can take advantage of already compiled lists of most common hidden resources.

[10] See https://www.jetbrains.com/

Certain server misconfiguration can make locating of the hidden resources even more trivial. When the server has enabled so called *directory browse*, accessing the existing folder without default file (`index.htm`, `default.aspx` and similar) will provide tester with a listing of files present in the directory [33].

As it was already mentioned, testing for hidden resources can be done in an automated manner without changing the target application's state.

### Application Finger-printing

When the target application runs on the well-known software solution like *Wordpress*[11] or *Drupal*[12], or is powered by similarly known framework, such as *Laravel*[13] or *Zend*[14], it may be beneficial for the penetration tester to figure out what the application or the framework in use is. The process of figuring this type of information is known as *finger-printing* [27]. In the same way, servers that are physically hosting the target application can be also finger printed [27].

The benefit of discovering what application or framework is powering the target, on what server, lies in the opportunity to look up already known exploits that were developed for finger-printed applications, frameworks or servers.

Detection itself is usually based on the number of specific characteristics that the application, framework or server exhibits — to name a few, it may be specific *meta tag* inside the request response, *HTTP header* disclosing the server software version and name or existence of certain specific file [27].

Number of already implemented solutions for identifying technologies and software used in the web application exist — among the most notable ones are the *WhatCMS*[15] and *Wappalyzer*[16]. Both of these services offer paid API which makes integration with automated tool possible, but pricey. Difference worth mentioning is that when the third party tool such as any of the two mentioned is used, requests against the target application are coming from the different source and not the penetration tester's machine — this can be beneficial when some rate-limits for the testing session apply.

## 2.5 Existing Software for Penetration Testing

This section takes a closer look on already existing tools that are designed to be used for web application penetration testing. For each of the analyzed tools its intended purpose is stated as well as brief description of the tool's functionality, implementation language and target platforms. Detected strenghts, weaknesses and limitations are mentioned.

[11] See `https://wordpress.com/`
[12] See `https://www.drupal.org/`
[13] See `https://laravel.com/`
[14] See `https://framework.zend.com/`
[15] Tool is available online to try `https://whatcms.org/`
[16] Tool is available online to try `https://www.wappalyzer.com/`

### truffleHog

The *truffleHog*[17] is an utility written in Python language, designed for finding interesting strings committed into the git repositories. Such interesting string is often called a Secret. Secrets are published into the publicly available repositories most often by accident. For big organizations that maintain dozens, sometimes hundreds even, of repositories, it may be hard to efficiently keep the track of potentially sensitive information being published. And the *truffleHog* can help with just these efforts.

#### Identified Strengths

- Commit history awareness — *truffleHog* identifies secrets that were removed or rewritten in the past but were left in the commit history.
- Wide range of parameters — there is a great range of options that can be configured before launching the tool. Support for JSON output, possibility to selectively turn off some of its functions or specify maximum depth for searching the commit history.
- Possibility of extension — it is possible for a user to write their own rules to extend the search.

#### Identified Limitations

- Custom extension rules must be supplied as JSON — when the tool loads the rules from provided JSON, it naturally fails for invalid JSON files. This requires escaping certain special characters which appear in rules' definitions.
- Does not run on files or directory structures — it is not possible to use the *truffleHog* to scan single files or the directory structures without additional effort (e.g. setting up dummy repository in the directory structure).
- No easy way to detect short secrets — implementation at the time of writing this thesis does not allow searching for secrets shorten than 20 characters, as the high-entropy algorithm would start to pick up strings that are not real secrets.

The *truffleHog* tool with all its strenghts and weaknesses is the kind of tool that would fit into the software company's Blue Team repertoire, to help them keep an eye on both their public and private repositories.

Other similar tools exist and were researched in the course of working on the thesis. In their nature, they work similarly to *truffleHog*, and they are not getting dedicated section in the thesis. Instead, they are discussed briefly in the following paragraphs. Researched tools are *Reposcanner*[18], *repo-supervisor*[19] and *git-all-secrets*[20].

**Reposcanner** is a tool written in Python, very similar to the *truffleHog*, that brings a number of new useful parameters for the scan configuration. Parameter `-e/-entropy` allows configuration of the entropy level being reported, the `-c/-count` allows limiting

[17] Tool is available online: https://github.com/dxa4481/truffleHog
[18] Tool is available online: https://github.com/Dionach/reposcanner
[19] Tool is available online: https://github.com/auth0/repo-supervisor
[20] Tool is available online: https://github.com/anshumanbh/git-all-secrets

number of commits that are scanned, and `-l/-length` allows for limiting the maximum line length to be considered.

Another tool, **repo-supervisor**, this time written in *JavaScript* with *Node.js*, is built specifically for scanning of *JavaScript* and JSON files for secrets. Compared to the previous tools, it does not need to scan the git repository to work, it can scan directories and single files as well. This option makes it possibly a great tool to be used in automation scenarios.

Example of such scenario could be chaining it together with another tool, that creates local copy of the files used by target application. After the tool creates the local copy, the *repo-supervisor* is launched on the local copy directory and the secrets are provided on the output.

Another great functionality provided by the *repo-supervisor* is the possibility of extension of the supported file types, as described by the *repo-supervisor* author in documentation [17].

Last researched tool of this nature is a **git-all-secrets** utility. It is written in *Go* language and already utilizes other existing tools to search for the secrets. It uses both the *truffleHog* and *repo-supervisor* and adds great variety of run parameters on top of them.

The tool is again focused on retrieving secrets from git repositories with support for organization-wide and enterprise-wide repository scanning, including: scanning all the public and private repositories belonging under *GitHub* organization, scanning all the repositories of users inside the *GitHub* organization, scanning the user gists and various other combinations of the mentioned at once. It is a "big gun" tool that is capable of in-depth search for secrets across the whole organization.

### waybackurls

There are multiple implementations in various programming languages, from different authors, that exhibit the same functionality. This section presents the research of the two such tools. The first one is written in Python[21], the other one is written in Go[22] language.

The *waybackurls* tool takes advantage of Internet Archive feature that allows extracting the historically recorded url addresses of given domain or domains. It does that using Internet Archive's search feature to search for regular expression which matches any possible address on given domain or its subdomains. Internet Archive offers the `GET` endpoint for this search (the `$DOMAIN$` is to be replaced by the domain of interest), as shown in figure 2.2.

```
http://web.archive.org/cdx/search/cdx?url=*.$DOMAIN$/*&output=json
&fl=original&collapse=urlkey
```

Figure 2.2: Web Archive endpoint that allows searching for the existing records of specific URL in the index.

[21] Tool is available online: https://gist.github.com/mhmdiaa/adf6bff70142e5091792841d4b372050
[22] Tool is available online: https://github.com/tomnomnom/waybackurls

#### Identified Strengths

- No interaction with target application — The tool retrieves known URLs without direct interaction with the target application, because it is requesting the data from *Internet Archive* service.
- Potential of hidden address discovery — As *Internet Archive* keeps snapshots of historical versions of the pages, it is possible to retrieve addresses that were later hidden by target application authors, but are still accessible. This may be also beneficial for the penetration tester, because they can learn about the way old URL parameters were named and chained, which can help them to predict or understand current application design better.

#### Identified Limitations

- Incomplete sitemap — It is not guaranteed that every page of the target application is known to *Internet Archive* and therefore returned results cannot be considered a complete map of the target application.
- Possibility of a significant noise overhead — The potential for discovering hidden addresses is sadly a double-edged sword. Should the target application undergo a lot of version changes throughout its history, and should there be snapshots of these, great number of returned results will be completely invalid and irrelevant.

Employing this tool or its functionality in the penetration testing automation process could be very beneficial, especially in the context of this work. Not only the interactions used to collect information about target web application are non-destructive and non-permanent, they are also undetectable by the target applications, as none of these interactions is executed directly towards the target application.

Penetration tester can gain a lot of knowledge about the target application without sending single request against it. They can discover what URL parameters are or were used by the application, how application's directory structure is roughly organized, what subdomains of the target application were used in the past and probably much more. Disadvantages exists though — as the *Internet Archive* stores snapshots of webpages from points in history, it is possible for the retrieved information to be be out of date and/or no longer relevant.

### Arjun

*Arjun*[23] is a tool designed to discover hidden URL parameters that are available on supplied URL address. It is written in *Python* language, for 3.4 vesion and higher.

Available parameters are discovered using two techniques — looking for HTML attributes that could be linked to parameter name and brute-forcing the parameters based on the list of well-known URL parameters. First, the target application's URL without any added parameter is requested and the response is recorded. After that, six characters long pseudo-random string is generated and used as dummy parameter for the target URL (it is expected

[23]Tool available online: https://github.com/s0md3v/Arjun

that this parameter does not exist on target application), new request is sent and the response is again analyzed.

During the first response analysis, HTML elements such as `form` or `input` are looked for and their attributes are parsed for possible parameter names. If the parameters are found they are added to the list of parameters which will be later fuzzed by *Arjun*.

While analyzing the second request, application checks whether pseudo-randomly generated string used as a parameter exists and whether its name or value were reflected in the response. Both requests' response codes are compared. This way the *Arjun* learns how target application reacts when being provided with parameter that does not exist.

Once the initial requests are sent and analyzed, *Arjun* starts preparing and sending out new requests to the target application's URL with appended parameters based both on the list of well-known URL parameters, and the possible parameters extracted from the first request's response.

**Identified Strengths**

- Support for `POST` and `GET` HTTP methods — Launch parameters allow penetration tester to specify what HTTP method the tool should use when looking for hidden parameters.
- Possibility to supply custom URL parameter lists — Penetration tester can supply their own list of parameters to check.
- Multi-thread implementation — Launch parameters allow to specify number of threads to be used when sending out HTTP requests.
- Spot on launch parameter options — Parameters to set specific headers for requests and to set a delay between requests.

**Identified Limitations**

- Header specification possible only through an external editor — Already mentioned setting of request headers is possible, but text editor is required to do so. Possibility to specify headers directly from command line would be more convenient and flexible.

Again, similar tools like *Arjun* exist and they will be briefly discussed in the following paragraphs, the tools are: *param-miner*[24] and *parameth*[25].

The *parameth* is simply another implementation of the similar principles that were implemented in *Arjun* utility. It is described as a tool to brute discover POST and GET parameters [24] and provides few additional parameters to extend the functionality — like setting additional headers from command line, specifying the user agent or treshold difference of the initial response and the response with new parameter, and few more.

The *param-miner*, on the other hand, is a *Java* plugin for BurpSuite that provides wide variety of settings that can be configured. It searches for hidden URL parameters and also HTTP headers with configurable possiblity of enabling brute-force. Other configurable

[24]Tool is available online: https://github.com/PortSwigger/param-miner
[25]Tool is available online: https://github.com/maK-/parameth

settings include, but are not limited to: setting number of threads to be used, discovering *cache poisoning*, searching only for the URL parameters or the headers, choosing between different brute-force wordlists, and a lot more.

## 2.6 Testing Web Application Sets

In this section, the selection of suitable testing set of Web Applications to test against is covered. Basic selection criteria are proposed and based on them, the two testing groups of Web Applications are created, each with different purpose.

### Approach to Test Set Selection

To test the implementation of the tool from practical part of the thesis, a testing set of web applications must be established. Author proposes creation of the two distinct groups of web applications:

- Highly secure web applications — applications that are often subjected to penetration testing and therefore expected to maintain higher level of security.
- Random applications or applications that are not frequently penetration tested — applications that are subjected to either very rare penetration testing or no penetration testing at all.

Web applications falling down into the first group are defined by their expected level of security. Author of this thesis expects that testing the tool against the first group testing set — highly secure web applications — will yield little to no positive results, as the web applications from this group are tested on daily basis by other penetration testers.

Testing against the second group testing set — random or under-tested applications — should on the contrary, yield some amount of positive results. Testing against these groups and observing results during implementation could help to verify that implemented tool operates properly.

In the final testing stage, the tool will be tested against both of these groups. The final testing set should be comprised of applications with various levels of security. Results of the final testing will then say how successful the implementation and the approach were.

### Test Set 1: Highly Secure Web Applications

To put together a list of highly secure web applications, the two most known bug bounty programme sites were used — *HackerOne.com*[26] and *Bugcrowd.com*[27]. These sites provide way for penetration testers and hackers to safely test real web applications for vulnerabilities, while providing companies with an opportunity to get their applications tested. As these websites are frequented by a lot of penetration testers, web applications put up there are expected to be thoroughly tested and highly secure.

[26] See https://hackerone.com
[27] See https://bugcrowd.com

Other benefit of using only applications from the bug bounty programmes is the implicit consent to penetration testing as long as given bug bounty programme rules are followed by the testers.

Set of expectedly highly-secure web applications is presented in table 2.1.

| Company | URL | Notes |
|---|---|---|
| *Flickr* | https://flickr.com | And all its subdomains. |
| *BOHEMIA INTERACTIVE* | https://bistudio.com<br>https://bohemia.net<br>https://ylands.com<br>https://ylands.net<br>https://arma2.com<br>https://arma3.com<br>https://dayz.com<br>https://armamobileops.com<br>https://minidayz.com<br>https://vigorgame.com | |
| *Hyatt Hotels* | https://www.hyatt.com<br>https://world.hyatt.com<br>https://starbucks.com | |

Table 2.1: Table is listing expectedly highly secure web applications, their URLs and the companies that run their bug bounty programmes are mentioned.

### Test Set 2: Under-tested Web Applications

To put together a list of questionably secure web applications, author browsed the Internet for websites that belongs to smaller to middle sized companies, bloggers and individuals that would be willing to offer their consent and let their web application get tested. Main criterion for selecting a company for this set was whether the company had website built on top of a professional CMS or whether someone put the website together from a scratch. Applications built from scratch were favored.

At the time of writing the thesis, author reached out to 50 companies out of which only one replied. Negatively.

The table 2.2 presents the set of questionably secure web applications. Owners and maintainers of these applications gave consent to penetration testing that is to be conducted in context of this thesis. Where applicable, specific circumstances of the given consent are mentioned in *Notes* column.

| URL | Notes |
|---|---|
| http://davidriha.cz | Author allowed penetration testing in exchange for final report being shared with them. |
| https://danieldusek.com | Thesis author's web application. |
| https://www.netsearch.cz/ | And their infrastructure. Permission acquired through the work's supervisor. |
| https://nesad.fit.vutbr.cz | Scope is limited to the resources maintained by the work's supervisor. |
| http://testphp.vulnweb.com | *Acunetix* scanner's test web application known to contains security issues. |

Table 2.2: Table is listing applications that are not frequently penetration tested.

# Chapter 3

# Proposed Approach

This chapter starts with the definition of *non-destructive* and *non-permanent* actions in the context of this thesis and their relationship with *HTTP* (protocol) and *HTTP requests*. After that high level approach to web application penetration testing proposal is presented, its phases are introduced and then explained on examples.

## 3.1 Non-destructive and Non-permanent Actions

The final product of this work aims to test and evaluate the security state of target web application, without changing or damaging it or the service it provides. Communication between the implemented tool and the target application occurs strictly over the HTTP, using the *HTTP requests*.

In the thesis the *non-destructive* and *non-permanent* interactions are considered to be the *HTTP requests* sent with so called *Safe Methods*, or subset of methods and parametrized methods of so called *Idempotent Methods* group. *Safe Methods* are described as essentially read-only methods where the client sending the request does neither expect nor requests change of the state or data on the remote server. These are the `GET` and `HEAD` methods [10].

Additionally, `OPTIONS` and `TRACE` methods should have no side effects [10] and in the context of this work will be considered *non-permanent* and *non-destructive*. The tool also takes advantage of the `POST` method with certain very specific parameters, with which the action is expected not to change the target application's state. There are no guarantees to support this use and it is discussed in the later chapters.

It is possible for any of the considered *non-destructive* and *non-permanent* interactions to be destructive when executed against the target application that is implemented in contradiction to HTTP specification. In such case, it is considered the resource owner's responsibility [10].

## 3.2 High-level Approach Proposal

This section proposes high level design of the generally applicable approach to web application penetration testing that is focused on retrieving information about the target

application solely by using non-permanent and non-destructive interactions as they are defined in section 3.1.

Proposed approach expects URL address to be initially the only input provided to the penetration tester. In the very first phase, a tester should collect as much openly available information, about the application residing on provided URL, as possible. Collected information should be then filtered, processed and categorized. Separate categories are use-case specific and are up to a penetration tester to properly establish them.

Based on the category to which collected information belongs, additional processing steps, again, category specific, should be taken. Report summarizing the findings will be then generated from the results of steps taken and provided back to the penetration tester. Depending on the nature of information collected and processed, the output report can take form of discovered vulnerabilities, suggested actions or something else entirely.

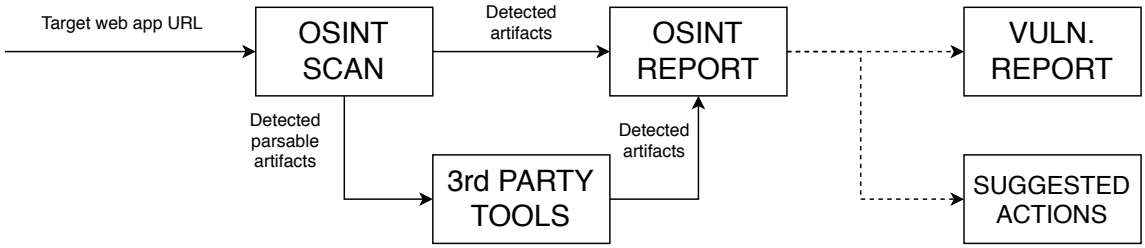


Figure 3.1: Block scheme of the process flow describing the high-level proposal of the generally applicable approach to web application penetration testing. Flow starts with the input consisting of the url address of the target web application heading towards the *OSINT SCAN* block. Once the OSINt scan is performed, two sets of detected artifacts are put on the output, both heading to different blocks. *Detected artifacts* that are no longer parsable go directly into the *OSINT REPORT* block, while *detected parsable artifacts* are first processed by other, possibly 3rd party toolsets (visualized as *3rd PARTY TOOLS* block). After parsible artifacts are processed, they also flow into the *OSINT REPORT* block. Two expected forms of output from *OSINT REPORT* block are expected, the *VULNERABILITY REPORT* containing the specific vulnerabilities that are present in the target web application, and the *SUGGESTED ACTIONS* containing recommendations for a tester regarding further penetration testing.

## 3.3 Pre-penetration Testing Considerations

For legal reasons, it is not possible to just execute penetration testing against any web application — under the *Computer Fraud and Abuse Act (18 U.S.C. 1030)* it is considered a federal crime to "intentionally access a computer without authorization or exceed authorized access". The mentioned statement is very broad and can be applied to almost any form of penetration testing and can lead to penetration tester being prosecuted.

It is advisable for the penetration tester to acquire a written consent with penetration testing, or to test only web applications that clearly state their penetration testing policies, and adhering to these policies while conducting the testing. At the time of writing the thesis, a `security.txt` Internet-Draft is being developed with one of its goal being the standardized indication of security policies availability [11].

While it should not be possible to damage or irrecoverably change properly implemented target application, the **risk of overloading** it must be considered. Most of the information about the target application will be expectedly retrieved by issuing *HTTP requests* against it and by reading the response.

Before the penetration tester engages in automated testing, they should consider what rate-limits are to be enabled on the tool to prevent accidental overloading of the target. This information can be sometimes extracted from the *rules of engagement* document, if the target application provides one.

## 3.4 OSINT SCAN Phase

The OSINT SCAN block represents the first and the most complex phase of the proposed penetration testing approach. A tester sets up areas of interest and their requirements, researches and implements tools to scan these areas in an automated manner and runs the scan. The first phase of the approach is proposed to be split into the following, chronologically ordered steps:

- **Define** — Tester decides and defines which areas are to be scanned on the target application and what form will the results have. For example, a tester may be interested in acquiring the complete site-map of the application, either because:

  (a) formal requirements on the final penetration testing report require complete site-map to be present in the report, or,

  (b) tester intends to run other tools against the discovered URLs.

  In case of (a), the site-map is considered to be a *non-parsable artifact* which is passed directly to the *OSINT REPORT* phase where it later gets turned into the part of the final report. In case of (b), the site-map is considered to be a *parsable artifact* which is fed into one or more, possibly 3rd party, tools, that will further process it. Only after the parsable artifacts are processed and made into non-parsable artifacts, they are again inputted into the *OSINT REPORT* phase.

- **Research** — With areas and requirements from the previous step in mind, a tester researches existing tools that could be used for obtaining the artifacts in an automated manner. License limitations of the researched tools must be taken into an account as well as testing security considerations (see section 3.3) and the nature of interactions executed by the tools (see section 3.1). Using the previous example, penetration tester is supposed to discard using the site-mapping tool that would submit forms on the target application when mapping the it.

- **Implement** — After tester identifies which parts of the OSINT scanning process can be solved by using the 3rd party tools and which parts of the system need to be implemented from scratch, they design and implement the system that is capable of conducting the scan and providing corresponding artifacts to the following phases. While designing and implementing it may be desirable to consider future use of the tools in the same context and account for portability in advance.

- **Run** — Tester runs the implemented system and executes the automated penetration testing. Run outputs (the artifacts) are passed onto the next phases of the penetration testing approach.

### Direct and Indirect Requests

There are services like *Google Search* or *Web Archive* that make a copy of the website in certain point in time, and some of these services offer API for searching in their copies. Through these APIs it is possible to discover information about the target application without even directly requesting it. Taking advantage of such services can prove itself helpful when expected number of requests on the application is high and rate-limits apply.

Similarly, when application or its parts are hosted in the publicly available repository (*GitHub*, *GitLab* and others), analyzing application source code is possible without the direct interaction.

Author of this thesis proposes considering use of these and similar services in the **Define** step of the OSINT SCAN phase — both to broaden the scope of the testing and the amount of information processed.

### Side Channel Monitoring

Author of this thesis also proposes an extension to be considered in the **Define** step, when the tested application belongs to a larger organization with many employees. The idea of this extension is to identify channels that are used by the employees (or sometimes even customers) of the organization and can be monitored for information relevant to the scope of the penetration testing. For example when developers troubleshoot their code and share it via third party channels like *Pastebin*[1], fragments of sensitive and valuable information may be leaked.

Identifying these channels and setting up their automated monitoring — based for example on detecting certain sensitive organization-related words — can provide a tester with valuable information which can be further used for penetrating the application. The automation here is crucial as it allows the processing of high volume of information, that would not be otherwise possible.

## 3.5 3rd PARTY TOOLS Phase

One of the goals of the proposed approach is to enable automated execution of penetration testing and to allow processing of high volume of information that would not be humanly possible. For this very reason, the approach contains the *3rd PARTY TOOLS* phase which is designated to use of already existing and developed tools for additional data processing or penetration testing tasks.

Few examples of additional data processing that could be applied through the use of third party tools to enhance the testing results:

[1] See https://pastebin.com

- *Machine learning algorithms to classify themes or topics of analyzed pages* — In the specific scenario when penetration tester wants to brute-force user password, understanding the general content of the page could be beneficial for targeted dictionary generation.
- *Extracting contact information* — Penetration tester can be tasked with assessing what contact information is disclosed on the target application. Extracting emails and phone numbers from the textual response is to be offloaded to the third party tool dedicated for this purpose.
- *Static analysis* — External application for performing static analysis on JavaScript source files can be used to discover secrets or application endpoints.

This phase should not be understood in a sense that *exclusively and only* the 3rd party tools may be used — a tester can, of course, decide to reuse their own tools to evaluate the data or to further test the application.

## 3.6 OSINT REPORT Phase

The phase is dedicated to working with gathered non-parsable artifacts coming both from OSINT SCAN and 3rd PARTY TOOLS phases, and compiling them into the final testing report. Output of this phase may be influenced by the requirements on the penetration testing as specified by the party that is being tested. In general, the output can be expected to take form of a report of discovered vulnerabilities, a list of sensitive information or information worth inspecting, a list of recommended steps to be taken by a penetration tester before finishing up the penetration testing session, or a presentable document with testing results.

For illustration, few examples of the output artifacts:

- Discovered application secrets or access tokens,
- complete site-map,
- list of discovered resources,
- detected vulnerable endpoints,
- endpoints suggested for further testing,
- leaking personal information, or,
- system users with weak or well-known passwords.

Of course, as the output of the *OSINT REPORT* phase can take multiple forms — including a list of recommended actions. It makes sense to optionally include a virtual phase or block that would be dedicated to manual penetration testing where a tester acts on the recommendations provided by the tool.

# Chapter 4

# Tool Design and Implementation

In this chapter, *ReconJay* — a tool, that follows the approach proposed in chapter 3 — is presented. Early parts of this chapter describe how the tool is designed and structured and how the implemented tool maps to the aforementioned approach. Later parts of this chapter then present how were the main parts of the tool implemented and what are their capabilities and limitations.

## Technology and Platform Support

*Python* in version 3.6 was chosen as an implementation language for the *ReconJay* tool. The decision to use *Python* as the main implementation language was made due to its multi-platform portability [38] and available convenient packaging system *PIP*[1] that makes it easy to take advantage of third-party libraries [4] (such as the *requests*[2] and *beautifulsoup*[3] packages), hence leading to less wheel-reinvention situations and more clean implementation.

The implemented tool is intended to be run from the command line and does not provide any other than the textual interface. During its run, only the most important messages are written on the standard output, typically informing about the problems that arose or about the transitions between modules that took place.

An important part of the implemented tool is the presentation of the results. Once the tool finishes its penetration testing activities, a final report file is generated. *HTML*, *CSS*, and *JavaScript* technologies are utilized to provide this report. The web technologies were chosen for presentation purposes due to today's widespread availability of the software for viewing web pages, and due to a variety of formatting options brought by the *HTML* and *CSS*.

From the platform-support-wise perspective, the tool is designed and tested to operate correctly under both *Windows* and *Linux* based systems. While it was not explicitly tested to work on the *MacOS* operating system, it should be also possible to run the implemented tool there, as long as the *Python 3.6* or above environment is installed.

[1] Package Installer for Python: https://pypi.org/project/pip/
[2] Python Requests Package Homepage: https://python-requests.org
[3] BeautifulSoup Package Homepage: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

## 4.1 Relationship Between Proposed Approach and ReconJay

The *ReconJay* tool implemented in the course of working on this thesis aims to map the proposed approach to the concrete implementation. Purpose of this mapping is to showcase and demonstrate a real-life application of the approach in a practical implementation, that benefits from the qualities brought to the table by the approach.

In order to reflect qualities of the proposed approach, such as high re-usability and support for automation of the manual and repetitive tasks, most of the *ReconJay* tool's functionality is implemented as single-purpose modules, that are launched by a module loader. More on this later, in section 4.2 *Modules and Modules Launching Design*.

Initially, *ReconJay* receives only a target application's URL and launches the first of its modules, with the provided URL being its only parameter. This behavior directly maps to the beginning of the *OSINT SCAN* and *3rd PARTY TOOLS* phases (described in chapter 3). Note that the tool respects the number of initial parameters on the input. In the same way as the earlier described approach, it starts only with the target application's URL. Figure 4.1 illustrates the mapping between approach phases and the *ReconJay* tool's behavior.

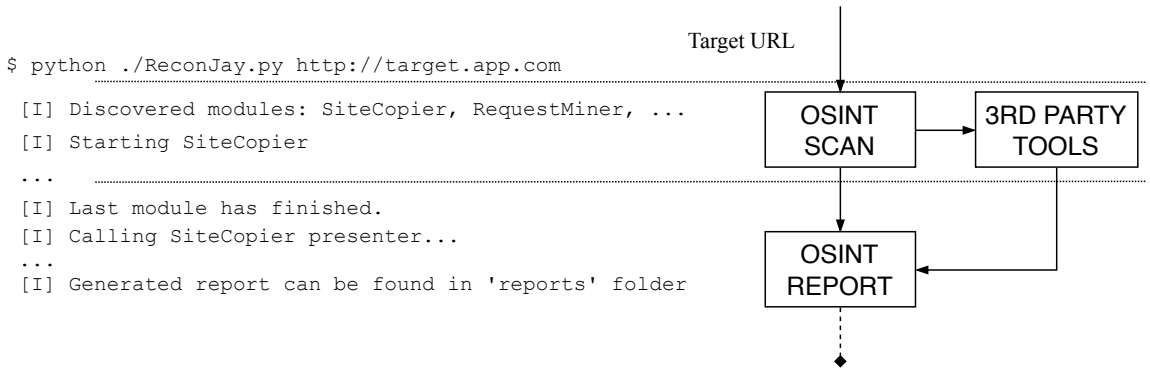


Figure 4.1: Mapping of the *ReconJay* tool's execution stages to approach proposed in the chapter 3.

The proposed approach mentions *artifacts* and *parsable artifacts* as an output of *OSINT SCAN* and *3rd PARTY TOOLS* phases. Again, this maps directly into the *ReconJay* tool's behavior. When launched module returns its results back they are expected to be either *parsable* or *non-parsable*. Based on their type, they are further piped as an input into one or more other modules, or into the *OSINT REPORT* phase.

As was already hinted in the previous paragraph, the output from the modules that were run during the tool's execution is used for report generation purposes. This functionality maps to the *OSINT REPORT* phase of the approach. The *detected artifacts* on the report generation phase's input map to *non-parsable* results recorded by the launched modules.

## 4.2 Modules and Module Launching Design

The final product of this thesis is split into two main parts — a module loader and modules facilitating the actual web application penetration testing. Its intended purpose is to sup-

port higher re-usability and healthy responsibility separation, both of which are considered to be the key principles of good software design [22].

The first part of the solution — a module loader — is focused on discovering implemented modules and taking the necessary steps to successfully execute them. This process encompasses:

1. Discovering implemented modules,
2. determining an order in which the modules are to be executed,
3. collecting module results and enabling inter-module communication,
4. generating a report.

Figure 4.2 illustrates the decision process leading either to module execution or to the application's premature termination.

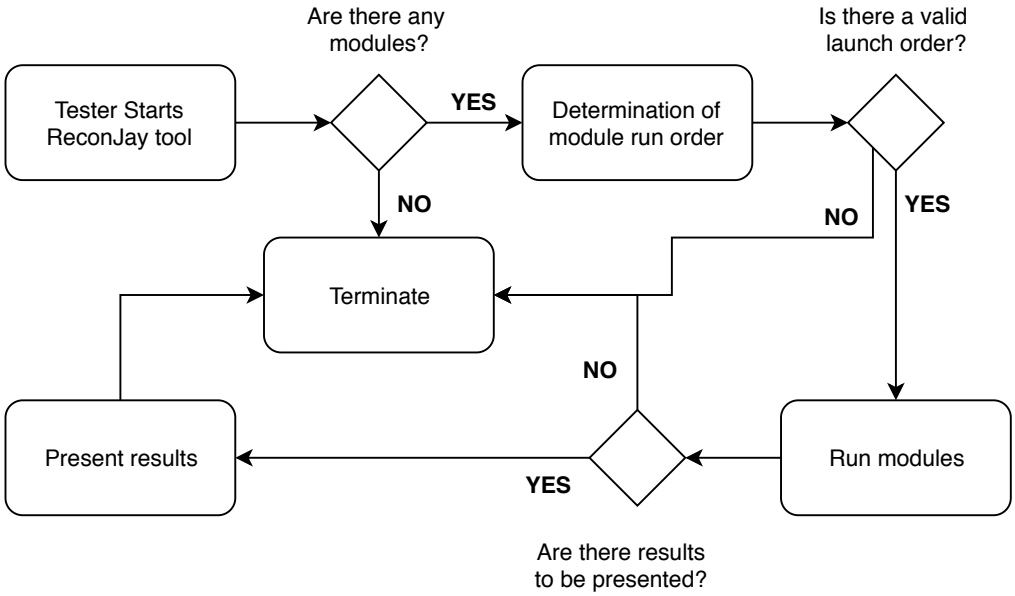


Figure 4.2: Conceptual diagram of the module-loading process flow. There are three decision nodes between the start and the end block: Are there any modules? Is there a valid launch order? Are there results to be presented? Whenever an answer is no, the process terminates. In case of positive answers, process flows from a tester starting the *ReconJay* tool, through determination of module run order and running the modules, to results presentation and consequent termination.

The second part of the solution — implemented modules — is essentially the most important part of the final product in terms of actual penetration testing. Each module encases concrete security scanning functionality with its specific goal or bulks of such functionality that cover related domains of the web application security. Additionally, each module is equipped with a presentation class that contains logic for presenting a module's results.

In a real-life scenario, penetration tester is expected to keep their own *modules* folder with all the modules they have historically implemented for various penetration testing sessions. Based on the requirements of future testing sessions, they would then only need to pick which modules apply for the target application at hand and use them.

### Module API

There are certain API methods that every module needs to provide, in order for it to be loaded by the module loader. This section lists the methods:

1. `get_dependecies()` — Returns a dependency object (see *Module Dependencies* section 4.2 later in this chapter) that is used by the module loader for module run scheduling purposes.
2. `set_options()` — Allows to set a module's run options from the module loader.
3. `provide_results()` — Allows a module to access results of other modules. Every module keeps a copy of results of the modules it is dependent on.
4. `execute(target)` — Called by the module loader when a module is supposed to execute the security-related functionality it addresses. Target application's URL is passed through the descriptively named parameter.
5. `get_presenter(results_structure)` — Retrieves an instance of the `Presenter` class that belongs to a given module. The `results_structure` parameter makes available the results of all the modules that were run during the *ReconJay*'s execution.
6. `get_results()` — Provides module artifacts back to the module loader to be shared (see *Module Results Structure* section 4.2 later in this chapter).
7. `leaves_physical_artifacts()` — Provides information about whether a module leaves physical artifacts after it is run (e.g. files in the file-system).

As long as the aforementioned methods are provided, a module can contain any number of other methods and reference any number of other classes it needs to perform its job.

### Module Results Structure

After a module finishes its work, the module loader accesses its results via the `get_results()` method call. To pass results between the module loader and even between other modules, a results object is used. This object reflects the possibility of returning *parsable* and *non-parsable* artifacts.

Code snippet 4.1 illustrates the expected format of the results object. Each module is expected to provide a single results object.

### Module Dependencies

Each module is capable of announcing its dependency on another module or modules prior to its execution. The module loader reacts to the announcement by scheduling a given module to be run only after their dependency module or modules has finished. This feature allows modules to build upon and extend on the results provided by other modules. It also

```
results_object = {
    "nonparsable": {
        "nonparsable_artifacts_01": [value, value2, ...],
        "nonparsable_artifacts_02": [value, value2, ...],
        ...
    },
    "parsable": {
        "parsable_artifacts_01": [ {objectX1}, {objectX2}, ...]
        "parsable_artifacts_02": [ {objectY1}, {objectY2}, ...]
        ...
    }
}
```

Code snippet 4.1: Results object provided from a module to the module loader upon `get_results()` method call. It contains two main properties — `nonparsable` and `parsable` which map onto the parsable and non-parsable artifacts proposed in the chapter 3.

enables a penetration tester to create more complex processing pipelines without breaking SOLID principles and without unnecessary redundancy in code.

Code snippet 4.2 illustrates the expected format of a module dependency object. Each module can provide a list of dependency objects to specify:

- What module it depends on.
- What is the type of the dependency (e.g. it is dependent on the output generated by the dependency module, or it only needs it to execute prior to itself).
- If the dependency is essential for the module to run.

```
dependency_object = {
    "depends_on": "DependencyModuleName",
    "dependency_type": "output",
    "is_essential": True
}
```

Code snippet 4.2: Accepted format of a dependency object provided by a module upon `get_dependencies()` method call. Each of the dependency object records contains information about a module it depends on, its dependency type (e.g. output, or just run) and whether this dependency is essential for a module to run.

### Module Options

The implemented tool supports functionality for adjusting configuration on a per-module basis. In the root directory of the tool's implementation, there is an `options.json` file which enables a penetration tester to configure specific settings of existing modules.

An option of a module is represented internally as a class property. Each module has its own options, which it allows to set — this is realized through the API method `set_options()`

and the property setting is based on white-listing those that can be set. The white-listing is done inside the mentioned method and it is done to avoid unwanted property overrides.

Code snippet 4.3 describes a structure of the `options.json` file, which is comprised of collections of options for every module.

```
{
    "ModuleName": {
        "DELAY": 1,
        "RANDOMIZE_SELECTION": True
    },
    "DifferentModule": {
        "MAX_REQUESTS": 1000,
        "DELAY": 5
    }
}
```

Code snippet 4.3: Structure of `options.json` file.

## 4.3 Results Presentation Design

Once all modules finish and their results are collected by the module loader, a report of the modules' findings needs to be generated — this corresponds to the *OSINT REPORT* phase of the proposed approach and provides a penetration tester with the means to review findings in a structured and readable way.

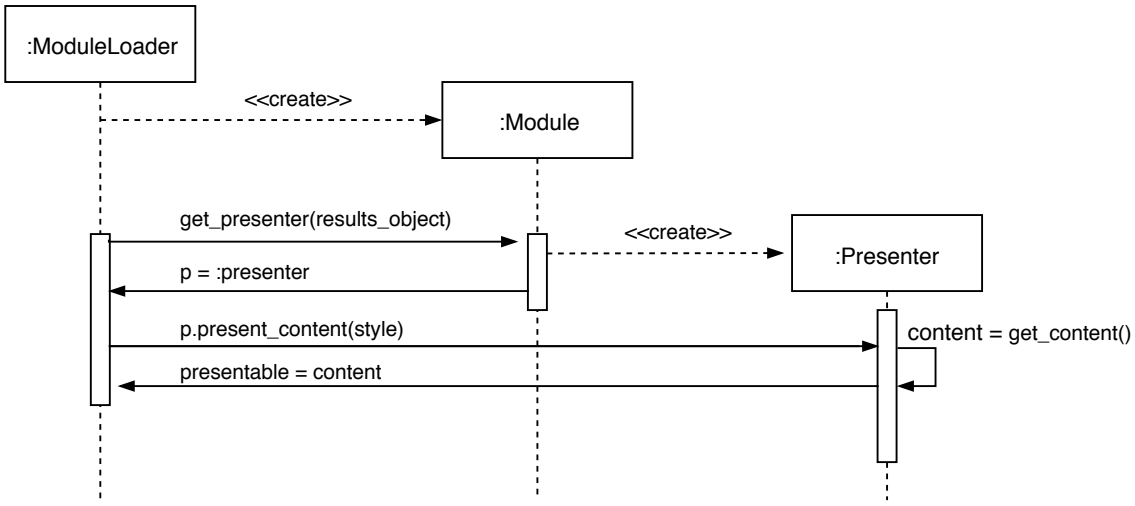


Figure 4.3: Interaction diagram featuring the `ModuleLoader`, `Module` and `Presenter` objects. First, the module loader instantiates a module object and invokes its `get_presenter()` method, passing it the results object that contains results to be presented. Module object then instantiates its presenter class and returns it back to the module loader. Finally, the module loader calls `present_content()` method on the presenter object and retrieves presentable data from it.

The responsibility for presentation and report generation is distributed among the modules. Each module has its own `Presenter` class that contains logic for presenting its findings

in a human-readable manner. When a `Presenter` class is indirectly instantiated from the module loader, it receives information about a preferred style in which the output should be generated. If a requested style is not available, `Presenter` class is required to fallback to default *plaintext* style.

Figure 4.3 illustrates the conceptual interaction between the module loader, a module and its `Presenter` class.

### Presenter API

In order for the module loader to be able to compile a final report file together from the partial outputs received from modules, it needs to interact with their `Presenter` classes in a standardized way. This section lists the API methods that need to be provided by a module's `Presenter` class:

1. `present_content(presentation_style)` — Returns content ready for presentation in the style specified by its parameter.
2. `generates_media()` — Returns information about whether supplementary media resources were generated as a part of the findings presentation (e.g. an image file or a proof of concept code).
3. `get_media_path()` — Returns path to directory in which generated media files are stored.
4. `get_importance()` — Returns information about how "important" a module perceives itself to be.
5. `set_information_level()` — Influences what type of information should be contained within a generated output.

## 4.4 Module Launching Implementation

An entry point into the implemented tool is located in the `ReconJay.py` file, which is in its essence the module loader. Its job is to discover existing modules and launch them — if they are correctly implemented, that is.

The module loader expects, by default, that all modules are stored under `modules` folder, which is located in the same directory as `ReconJay.py` file. The location of a module source folder can be easily altered.

Figure 4.4 presents a simplified directory structure of the implemented tool, focused on `modules` folder and the folders otherwise relevant to the module loading and launching. From the scoped directory `modules` it is visible that each module has its own directory with the same name as the module. Inside this directory, again a python file named after the module (in this case `SiteCopier`) is located. A presence of `__init__.py` file indicates that any *ReconJay* module is considered to be a python language module as well.

Folder `output` is used by the modules and the module loader to store artifacts produced during the application's run-time. With every run of the *ReconJay* tool, a new, time-stamped

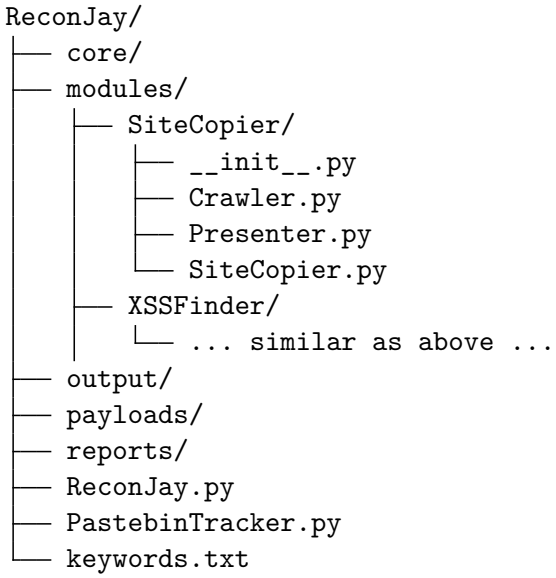


Figure 4.4: Tool's directory structure overview, scoped onto the *modules* folder.

and pseudo-randomly named folder is created within the `output` directory (e.g. for a *ReconJay* run initiated on 28th April 2019 at 1300 sharp, a folder `2019-04-28_130000_YG0GZ` would be created). Inside this folder, physical artifacts left behind by the modules are stored.

The `reports` directory serves for storage of the generated reports and is discussed later in section 4.5.

### Determining Launching Order of Modules

*ReconJay*'s modules can be dependent one on another and the proper order of their execution must be ensured. To guarantee that every module that can be run will run, the following algorithm is applied:

1. Module loader scans the module source directory (`modules`) and instantiates every module it discovers.
2. Module loader calls a `get_dependencies()` method on all instances and classifies modules into the following three categories:
   (a) Independent modules — Modules that do not require any dependency.
   (b) Dependent modules — Modules that require one or more dependencies of existing and available modules.
   (c) Non-runnable modules — Modules that return dependencies on modules that do not exist.
3. Module loader executes all independent modules.
4. Module loader schedules the run of modules with dependencies that can be satisfied.

5. Module loader executes scheduled modules.

6. If in the previous step at least 1 module was executed AND there are dependent modules waiting for execution, go to step 4. Otherwise report a circular dependency for every remaining module and terminate.

## 4.5 Results Presentation Implementation

After all modules finish, their recorded results are available in the module loader. The next step is to use these results to generate a human-readable report for a penetration tester.

The functionality responsible for the creation of the report lies within `PresentationHelper` class of the `core.helpers` module. This class provides functionality for compiling the final report together of all its parts, that are received from the corresponding modules. The `PresentationHelper` class keeps an internal representation of all the received parts of the report, where each part is comprised of **a module name**, **a module description**, **a formatted representation of the results** and **importance** of the part.

When the `generate_report()` function is later called, the mentioned importance is used to determine where in the final report should be the given part rendered — the higher the importance, the higher the position in the final report.

The `set_options()` method of the `PresentationHelper` class allows a tester to specify certain options with which the class will operate. The implemented tool features a boolean `show_module_description` option that controls whether a description of a module will be rendered into the final report. A possibility to suppress the module descriptions is a useful setting for scenarios when a penetration tester generates report for themselves — when they do not need to see the description, because they are the author — in contrast to when a penetration tester generates the report for someone else to review (e.g. for the requester of a penetration testing audit), who has no idea what which module does and why.

### Presentation Styles

The implemented tool supports two major styles of a produced report — a style named *BWFormal* (an abbreviation of "Black and White, Formal") and default "plaintext" style. See Appendix A.1 and Appendix A.2 for an example overview of these styles.

Each of the modules is responsible for its results presentation and supports at least the two mentioned styles. Decision what style will be used is made in the module loader part of the implementation and communicated to the corresponding presenters when they are called via the `present_content()` method. The method accepts a presentation style as its only parameter. If the requested style is not recognized by a presenter, it falls back to default plaintext style.

Outside the presentation of the results within a final report, two additional communication channels are utilized by the *ReconJay* tool to communicate with a tester — a terminal window and a verbose log file.

The former channel serves to communicate high priority status updates, messages and error reports to a tester that runs the tool. This is illustrated in figure 4.5

```
$ python ./ReconJay.py http://example.com
 [I] Run directory 2019-05-02_120229_XKC1R created.
 [D] Module classification done.
  Independent modules:
        |-> SiteCopier
  Potentially satisfiable modules:
        |-> MisconfChecker
        |-> RequestMiner
        |-> TokenFinder
        |-> XSSFinder
 [I] SiteCopier module output directory created.
 [SiteCopier]: Starting crawling operations...
 [SiteCopier]: Target acquired: http://example.com
 [SiteCopier]: 50 requests sent (Successfully: 50 | Failed 0)
 [SiteCopier]: 100 requests sent (Successfully: 100 | Failed 0)
 [SiteCopier]: 150 requests sent (Successfully: 150 | Failed 0)
 [SiteCopier]: TOTAL_REQUESTS_LIMITATION (150) reached. Crawling will stop.
 [SiteCopier]: Crawled work finished. Goodbye!
 [I] Module SiteCopier finished and saved results.
```

Figure 4.5: Illustration of a terminal high-priority communication channel.

The latter channel logs, in addition to the already mentioned type of messages, also the messages with a lower priority and of an informational character. In case of an unsuccessful application run, these messages can be used for debugging purposes and tracking down a cause of failure. Figure 4.6 illustrates this channel.

```
 [D] Discovered modules:
        |-> modules.MisconfChecker
        |-> modules.RequestMiner
        |-> modules.SiteCopier
        |-> modules.TokenFinder
        |-> modules.XSSFinder
 [SiteCopier]: Requested: https://danieldusek.com
 [SiteCopier]: DONE: 1 | QUEUED: 14 | FILTERED: 6 | FAILED: 0
 [SiteCopier]: Requested: https://danieldusek.com/.../GHMark.png
 [SiteCopier]: Response is binary, no links will be extracted.
 [SiteCopier]: DONE: 2 | QUEUED: 13 | FILTERED: 6 | FAILED: 0
 ...
```

Figure 4.6: Illustration of a verbose log file communication channel.

## 4.6 Modules

In this section, all of the implemented modules are first designed and then their implementation is discussed. The section covers modules responsible for crawling a target application and extracting information about it and modules that search for vulnerabilities and server misconfigurations. The closing paragraph of this section presents a standalone module for monitoring of the *Pastebin.com* side channel.

### SiteCopier Module

In the first stage of a penetration testing process, a target application needs to be explored and its content discovered. The *SiteCopier* takes on this task and provides its exploration results to the rest of the modules.

#### Design

*SiteCopier* module is designed to explore and crawl an application from its root URL address. In the process of doing so, it stores every request it sends and every response it receives from the target.

After retrieving response data for the first request, *SiteCopier* looks for and extracts the links and addresses contained in the response. Discovered links that are in the scope of a target application are normalized, duplicates are removed and then scheduled to be requested. Both request and response, including headers, are stored to disk.

Purpose of this module is not to verify a security standing of a target application, but rather to provide other modules with information about their target. Pre-fetching the content of an entire web application only once should also decrease a load on the application, and increase the speed of penetration testing conducted by other modules.

Interaction between *SiteCopier* module and its target uses only non-destructive and non-permanent HTTP verbs (as described in section 3.1) and ensures that a reasonable delay between consecutive requests is kept, to avoid target being overloaded. The delay can be configured by a tester.

It is possible for a web application to be composed of an infinite number of pages and *SiteCopier* addresses this possibility by the configurable limit on the number of requests that are sent.

#### Implementation

*SiteCopier* module uses two auxiliary classes to separate responsibility for crawling of a target application and for storing the tuple of a request, a response body and response headers onto the file system.

First class, `Crawler`, is responsible for crawling a target application. Crawling of available pages on a target application is achieved by sending pairs of HTTP HEAD and HTTP GET requests and processing received responses. Based on the content type of a received response, appropriate link extraction algorithm is used.

For *text/html* type of content, the HTML is parsed into the *beautifulsoup* object which is then queried for tags that may contain URL address in their attributes. Values of these attributes are extracted, URL addresses are normalized and duplicate entries are removed. Normalization of URLs consists of the following steps:

1. Removal of a trailing slash,
2. converting a host part and a scheme part to lowercase characters,
3. alphabetical ordering of query string parameters.

If any of the discovered URLs are relative, they are translated to their absolute versions before normalization process takes place. Responses of different content types are searched for URLs using a regular expression that matches http and https links.

Before a URL is added into the request queue, its fragment part is removed and it is verified that the URL was not requested yet and that it is not present in the queue already.

URLs that are out of the scope of a target application are filtered out and are not requested, the only exception being URLs of JavaScript or CSS files.

Respecting a delay between consecutive requests is ensured by sleeping down the application thread for the specified amount of seconds specified in the module run options (reflected by the `REQUEST_DELAY` property). In the same way, a maximum allowed number of requests is guaranteed.

Second class, `Storer`, is responsible for storing outgoing requests and incoming responses, their headers included.

By default, the requests and responses are stored under `output\run_folder\SiteCopier` directory (see section 4.4 for details on module run output directory naming convention). For every request sent, a new directory appears in that location, with the following three files:

1. `X.request` — contains target URL of the request, where X is the number of the request being sent (starts at 0).
2. `X.response` — contains response provided by a target application.
3. `X.response.headers` — contains headers that accompanied the response from a target.

Given the scenario when ten requests are sent, ten directories are created in the aforementioned location. The first named `0` and the last named `9`. Each of these directories will contain 3 files described above.

Figure 4.7 displays a relationship between the *SiteCopier* module and the two classes responsible for crawling the page and storing the requests data.

There is a limitation to the *SiteCopier* module's implementation — if a target application is serving a potentially infinite number of unique pages, the module can run in a loop infinitely too. A practical example of when this situation may occur is when there is

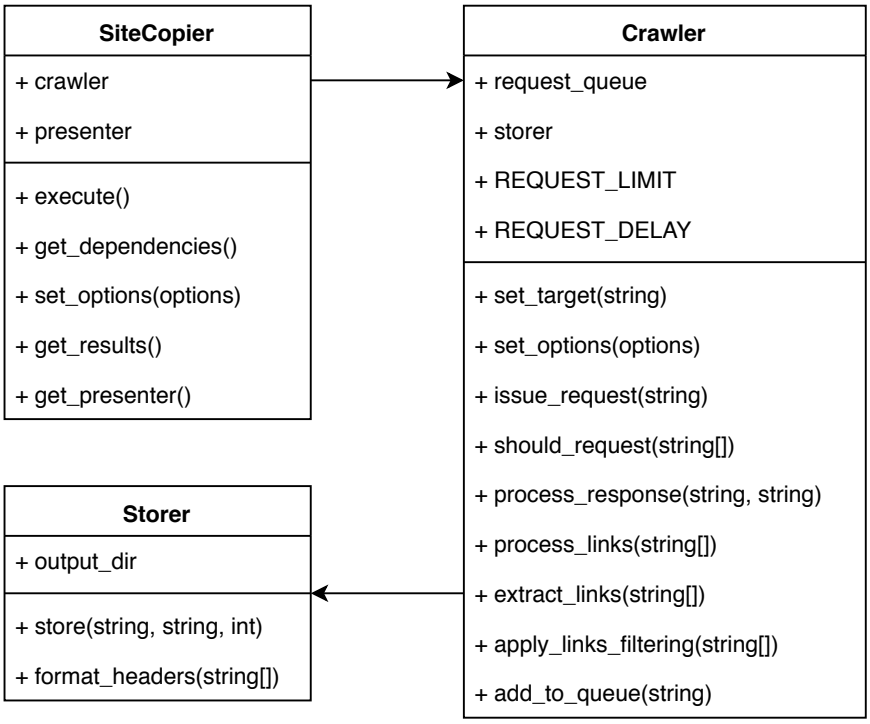


Figure 4.7: *SiteCopier* class diagram depicts relationship between the `SiteCopier` module class and two of its auxiliary classes, `Crawler` and `Storer`

a calendar component which takes the year number out of a query string. For every year it displays it also renders a link to the next and the previous year with an updated query string. These links are correctly extracted and processed by the module and scheduled to be requested, which is when the module also enters the loop.

To mitigate these scenarios, the *SiteCopier* module allows setting the `REQUEST_LIMIT` option prior to its run, to limit the maximum number of requests sent towards a target application. It is advisable for a tester to provide this option.

A secondary mitigation strategy to prevent *SiteCopier* from being stuck in a loop and requesting only the problematic part of a target is a pseudo-random request selection. Whenever a next request is being selected from the queue, it is randomly chosen by the `random.choice()` function provided by *Python*.

### TokenFinder Module

*TokenFinder* module serves for the detection of sensitive information contained in a target application's source code. The sensitive information is typically represented by high-entropy strings, which this module detects in responses provided by a target application.

### Design

The main purpose of the *TokenFinder* module is to locate the intentionally randomly generated strings, based on an assumption that such strings are secret and should not be disclosed to the public. Such strings can be detected due to a high-entropy [34].

These strings are likely to occur within the JavaScript application source code, or even inside HTML comments, and therefore there is no need for the module to interact with a target application directly. The *TokenFinder* module can use physical artifacts that the *SiteCopier* module collected and perform their inspection without directly interacting with a target.

The module is **essentially dependent** (see section 4.2) on the output of the *SiteCopier* module.

### Implementation

At the beginning of its run, the *TokenFinder* module reads all the physical artifacts collected by the *SiteCopier* module, which are stored in the module output directory. Then it starts iterating over them.

Every response recorded by the *SiteCopier* module is split by lines. The lines are then further tokenized into the tokens comprised only of characters that can appear in a valid *base64* string. There is also a requirement on a token's length — it must be over 20 characters long. It was experimentally tested that tokens shorter than 20 characters yield too many false positives when entropy is calculated. Figure 4.8 illustrates an example tokenization on a concrete line of code.

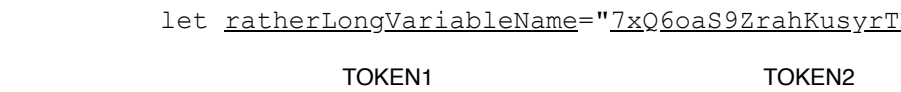


Figure 4.8: Example of line tokenization algorithm that splits the line of code into tokens of valid *base64* strings that are at least 20 characters long.

For each of these tokens, a Shannon entropy is computed using the algorithm depicted in code snippet 4.4. Calculated entropy value can range between 1.0 — the lowest entropy — and 8.0 — the highest entropy. If a token's entropy is higher than `ENTROPY_TRESHOLD` value, it is determined to be a potential secret. Its source address and position in the response is recorded.

```
def shannon_entropy(data, character_set):
    if not data:
        return 0
    entropy = 0
    for x in character_set:
        p_x = float(data.count(x))/len(data)
        if p_x > 0:
            entropy += - p_x*math.log(p_x, 2)
    return entropy
```

Code snippet 4.4: Algorithm used to compute Shannon entropy of the string.

### RequestMiner Module

*RequestMiner* module's intended purpose is to gather additional security-related information about a target application and provide this information to other modules.

#### Design

The *RequestMiner* module is capable of providing following information about a target web site:

- Existing query string parameters,
- existing response headers,
- hidden query string parameters,
- security standing of received response headers.

All of the mentioned above is detected through the use of non-destructive and non-permanent interactions (see section 3.1).

Existing values that are present on a target page, such as query string parameters or response headers, are parsed from physical artifacts recorded by the *SiteCopier* module.

Hidden values, on the other hand, are discovered by interacting with a target application in such a way, that their existence is verified. These values come from a source list of possible hidden values, that are tested to be present in a target application.

If a response to a request that was crafted to reveal hidden values is different from a response to a standard, non-altered request, it is considered a sign of hidden value or values being present.

A security standing of received response headers is evaluated by the third party tool designed for this purpose and the result is included in the report.

This module is **essentially dependent** on the output of the *SiteCopier* module.

#### Implementation

Discovery of existing query string parameters and existing response headers is done through iterating over physical artifacts left behind by *SiteCopier* module and extracting corresponding information.

For every discovered parameter, a parameter record is created or updated. Code snippet 4.5 shows the parameter record structure.

A list is used as a data structure to hold discovered existing response headers. Standard response headers, such as `Content-Type`, `Content-Length` or `Content-Encoding` are filtered out of this list for brevity and low informational value to a tester.

To discover query string parameters that are hidden — meaning there is no mention of them in an application's source code — requests to a target are necessary. The implemented tool

```
discovered_parameters = {
    "parameterName": {
        "sources": ['http://target.url/index.php?parameterName=1', ...],
        "values": [1, 2, 29, ...],
        "reflects": False
    },
    "parameterName2": {
        // Same as above
    }
}
```

Code snippet 4.5: A dictionary of discovered parameter names. For each parameter, a source URLs and discovered values are recorded. Additionally a check whether a parameter value reflects into the page is done.

uses a source file of 25 000 query string parameter names that most commonly appear in web applications.

Trying to discover hidden parameters contained in the source file one-by-one is not feasible for both time and traffic reasons. For this purpose, specifically crafted discovery URLs are prepared. Each of these URLs is kept below 2000 characters limit, and as many unique parameter names as possible are appended to them. For every parameter name, a unique, 8 characters long, pseudo-random string is generated (also called "canary") as its value. This reduces the number of requests that need to be made to discover hidden parameters significantly and also provides a mechanism to check for parameter reflection.

Prior to discovery URLs being requested, a pair of calibration requests is sent to determine:

(a) Target's response to request with a non-existing parameter,

(b) target's response to request without any additional parameters.

Once the tool starts requesting discovery URLs, it compares the received responses with the two previously mentioned responses. If the received response differs from them, the tool tries to pinpoint which of the included parameters in the discovery URL caused the response to change. When a canary string is present in the response, the tool detects it and immediately discovers which parameter or parameters exist. Additionally, such parameters are marked as *reflected*, as their values appeared in the response body.

When a canary string is not present in the received response, the tool tries the parameters contained in the discovery URL one-by-one.

A possible limitation to this approach is a situation when a target application reflects the whole URL or all of its parameters. This issue is mitigated by stopping the discovery process after more than 25 parameters are discovered to be reflected in the first received response. The tool then announces that probably every parameter reflects and proceeds to move onto the next tasks.

### XSSFinder Module

A subset of XSS, a *reflected XSS* vulnerability can be discovered in a target application through non-destructive and non-permanent interactions. Reflected XSS is typically based on displaying an improperly sanitized input provided by a user. This module aims to discover places where this vulnerability occurs.

#### Design

A data structure of reflected query string parameters acquired from the *RequestMiner* module is provided on the input of the *XSSFinder* module. The module then tries to discover and verify which of these parameters are vulnerable to reflected XSS vulnerability.

*XSSFinder* utilizes a *detector string* to understand how a target application sanitizes user input. The detector string is passed as a value for every of the reflected parameters and then a response is inspected and the level of user input sanitization is determined:

- **None** — User supplied input is not sanitized in any way.
- **Encoded for HTML body** — Characters < and > are encoded.
- **Encoded for attributes** — Single and double quotes are encoded.
- **Encoded** — User supplied input is properly encoded and all special characters (<, >, &, and single and double quotes) are encoded.
- **Otherwise Modified** — User input is not encoded in a way that can be considered safe, but also cannot be put into any of the categories above.

After input sanitization level is determined, the context of the reflected detector string is determined too. If no sanitization is performed, or if it does not correspond to the context in which the detector string is rendered, a potential XSS is reported.

When a sanitization level of *Otherwise Modified* is discovered, a potential XSS is always reported. This decision is based on an assumption that any other than complete output encoding is prone to vulnerability.

This module is **essentially dependent** on the *RequestMiner* module.

#### Implementation

At the beginning of the reflected XSS discovery, *XSSFinder* receives parameters that were detected as reflected by the *RequestMiner* and starts testing parameter after parameter on reflection, using their source URLs.

First, the detector string <"'> is used to determine a level of user input sanitization that is performed on the parameter value before it is reflected. If sanitization level is *None* or *Otherwise Modified*, a *discovered XSS object* is crafted and stored. If the sanitization level is *Encoded*, no further action is taken. Code snippet 4.6 shows a structure of the discovered XSS object.

```
discovered_xss = {
    "url": "http://target.app/index.php?page=Home",
    "param": "page",
    "protection": "None",
    "context": null
}
```

Code snippet 4.6: Discovered XSS object that is crafted for potential reflected XSS vulnerability findings.

When determined sanitization level is either *EncodedForHTML* or *EncodedForAttributes*, an attempt to guess its rendering context is made and if a mismatch between encoding style and rendering context is found, again, *discovered XSS object* is crafted and stored.

Context-guessing algorithm takes into account three non-whitespace characters to the left and to the right of the reflected detector string. If characters =, ", or ' are discovered in the correct order within this range, the *attribute* context is guessed. Otherwise defaults to guessing the *tag* context.

### MisconfChecker Module

This module's intended purpose is to detect a Version Control System (VCS) and server's Internet Information Services (IIS) misconfigurations that endanger deployed application.

#### Design

*MisconfChecker* module is capable of discovering certain improper IIS settings, accessible leftover version control files and enumerating hidden resources.

This module operates in two stages — during the first stage, hidden resources are enumerated and leftover VCS files are discovered, and in the second stage, an IIS misconfiguration, *enabled directory listing*, is searched for. The order of the stages has its purpose. Detection of the enabled directory listing is queued to be the second task so it can take advantage of discovered hidden resources and VCS leftovers.

Input for hidden resource enumeration and VCS leftover files discovery is a source file that contains common resources available in web applications, delimited by the newline character. Hidden resources include, but are not limited to, configs, database backups, and readable credentials files, as well as VCS configuration files.

Enabled directory listing detection accepts on its input a list of URLs that were seen to appear across the application. These URLs are tokenized into possible directory URLs and then requested.

The Provided response is checked to contain indicators of directory listing being enabled — if these are present, URL address exhibiting these characteristics is stored as a result.

### Implementation

Responsibility for the two stages in which this module operates is divided between `HRLocator` and `DLDetector` classes. The former class encases logic for enumeration of hidden resources and VCS leftovers, while the latter class focuses on detecting enabled directory listing. The relationship between the module and these classes is shown in Figure 4.9.

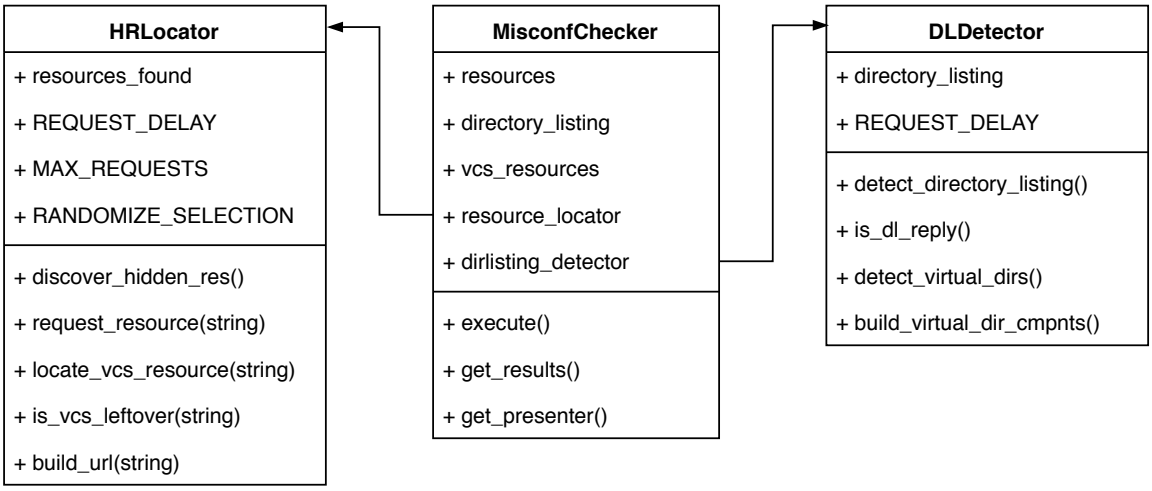


Figure 4.9: Relationship between *MisconfChecker* module and its `HRLocator` and `DLDetector` classes.

`HRLocator` first loads a list of resources to discover from the `resources.txt` file, which contains frequently used hidden resource names. The implemented tool provides a list of roughly 29 000 hidden resources to be discovered, but in a real scenario, a tester would use a resource list specific to technologies that are used by a tested application. The *MisconfChecker*'s resource enumeration capabilities are as effective as the list of resource names it uses.

A small subset of the `resources.txt` file is stored aside in `resources_vcs.txt`. This file contains commonly known VCS resource names and serves for the identification of VCS leftovers.

`HRLocator` builds and requests URLs of the hidden resources and checks responses returned by a target application. If any non-VCS resource returns 200 OK HTTP code, it is announced as a discovered hidden resource. When a VCS resource returns 200 OK or 403 Forbidden HTTP code, it is announced as a discovered VCS file.

Discovered hidden resource URLs are then passed to `DLDetector` that checks them for enabled directory listing. During its operation, it decomposes all valid URLs that were seen in the application into their probable directory paths and requests them. If the reply has an HTTP status code of 200 and there are enabled directory listing indicators present, the URL is added between the module results. Code snippet 4.7 shows strings that are used to indicate that a response was an enabled directory listing occurrence.

Once again, configuring the delay between consecutive requests and the total amount of requests that are sent is possible through module run options 4.2.

```
<a href="../">../</a>
<a href='../'>../</a>
<a href="/">Parent Directory</a>
<a href='/'>Parent Directory</a>
```

Code snippet 4.7: String indicators that if present in response from a server, indicate enabled directory listing feature.

### Standalone Tool: PastebinTracker

In some scenarios, it is possible to acquire sensitive information about a target application from channels that are not directly linked to it.

Crucial information may be shared by the users, employees or someone else entirely, **outside** the standard channels that are dedicated to it. One such channel is *Pastebin.com*[4], a website that allows users to share snippets of code — or more precisely, snippets of text. A typical use case — one user needs help with configuring a service they use and wants to share their current configuration. They decide to use *Pastebin.com* to paste their configuration and send a link *Pastebin* generates for them to a person that is helping them. The helping person even prefers this as the code will not get mangled up by the instant messaging they use, and if the configuration is written in a commonly known language, a syntax highlighting will be applied.

What they do not realize is that if they do not adjust visibility settings before posting, everything they just shared is public. The *PastebinTracker* tool that has been implemented in this thesis aims to take advantage of that.

#### Design

*PastebinTracker* tool detects and stores a local copy of interesting public pastes that are created by the users of *Pastebin.com* website. The interestingness of a given paste is measured by the presence of specific keywords in its contents. A tester can specify these keywords on the tool's input.

New pastes on the *Pastebin.com* website are created every few seconds and the tools reflect this by periodically fetching new pastes. Delay between consecutive requests can be specified by a tester.

When a paste that is processed contains one or more of the keywords specified on its input, its copy is created and stored for later inspection by a tester.

#### Implementation

*Pastebin.com* service offers a *scraping API*[5] intended to be used by the scripts and automated tools. For access to this API, a *Lifetime PRO* account is required.

[4] See online: https://pastebin.com
[5] See https://pastebin.com/scraping

After starting, the tool reads the keywords file that was provided by a tester. This file contains keywords which should trigger the tool to save a paste for further inspection. Keywords are separated by a line break.

The API is utilized in the *PastebinTracker* tool and it is periodically queried for new pastes. Right after the tool is launched, 20 latest pastes are fetched and searched for keywords specified by a tester. In a user-specified interval, the API is re-queried and new pastes are fetched and searched again.

When a paste contains keyword specified by a tester, its copy is created and written into a file. Each paste has a unique identifier which is used as a file name. Local copies are stored into the directory that a tester specifies by one of the tool's parameters.

## 4.7 Case Studies

This section presents two hypothetical case studies of the possible use of the *ReconJay* and *PastebinTracker* tools in two scenarios. The first case study deals with a penetration testing of a hobby web application smaller in size, with fewer daily visitors, while the second one deals with a penetration testing of a big company with thousands of employees. These case studies are presented to demonstrate the use of the implemented tools in different environments.

### Case Study I: Small Hobby Web Application

A male penetration tester Bob is tasked with penetration testing of a small hobby web site that belongs to a female owner Alice. From Alice, he knows the following:

- Web application uses *PHP* and *MySQL* technologies.
- Web application is built on top of the *WordPress* CMS.
- Web application has less than 100 accessible pages.
- Application does not use any 3rd party service that would require remote authentication.
- Hosting services are charged annually in advance and they are not derived from the application's traffic.
- Application owner prefers verbal explanation of possible discovered security shortcomings over the formal report.
- Target address of this application is `https://hobby.example.com`

Based on the acquired information, Bob briefly considers excluding the existing *TokenFinder* module from modules that are to be run but then decides against it to remain thorough.

As the application uses the *MySQL* technology, Bob decides to implement a new module called *SQLiFinder* that will detect error-based SQL injection in the application under

test. During its development, Bob follows all the coding best practices which in turn leads to the reusable new module being developed.

Knowing that the application is not charged per its traffic, Bob sets up an aggressive hidden resource enumeration. This resource discovery is conducted by the existing *MisconfChecker* module with no enforced `REQUEST_DELAY` value and no `MAX_REQUESTS` limitation. Bob is aware that the application is powered by *WordPress* CMS and so he prepares a resource enumeration file of commonly known *WordPress* and *WordPress Plugins* resources for the enumeration procedure.

Bob is informed about the application having under 100 accessible pages, so he preemptively sets up the limitation on 500 requests sent by the *SiteCopier* module. Bob sets this limit both to remain thorough in case the application has more than announced 100 pages, and to prevent the tool from getting stuck in case there is a crawling loop in the application (e.g. the calendar component with year reflected from query string parameter).

The application under test is a hobby page with few hundreds of daily visitors and no intention or support for building a community around it. Bob, therefore, decides not to monitor side-channels for information relevant to his target.

The last thing Bob does before launching the utility is to set the `show_module_description` presentation setting to `False` and request the tool to produce a *Plaintext* report. He will not be sending the report to Alice and the plaintext report is more convenient to work with from terminal.

Bob starts the *ReconJay* tool and after it finishes, he reviews the report. Discovered findings could be summarized as follows:

- *TokenFinder* did not identify any high-entropy strings contained in the application's source code.
- The total of 274 pages were crawled by the *SiteCopier* module.
- A number of hidden *WordPress*-specific query string parameter was discovered. Not a single of these parameters is reflected into the page.
- One non-standard *WordPress* parameter (`photoid`) was discovered. It is not reflected into the page.
- An occurrence of SQL injection was discovered in `photoid` parameter.
- No XSS occurrence were discovered.
- Enabled directory listing was discovered for a number of standard *WordPress* directories, such as `wp-includes` and similar.

Out of these findings, Bob draws the following conclusions:

- The application runs on a misconfigured server — the directory listing feature is enabled which allows attackers to enumerate files in known or discovered directories.
- A vulnerable photo-gallery plugin was installed on the *WordPress* instance that introduced a SQL Injection vulnerability.

Bob verifies these conclusions by an additional round of manual penetration testing (details of the manual testing are out of the scope of this case study). After verification of findings and conclusions, Bob sets up a meeting with Alice where he explains the results to her and offers further assistance in the remediation of the issues.

### Case Study II: Large Corporate Website

Once again, Bob, the penetration tester finds himself preparing for a penetration testing session. This time, he is expected to test a website that belongs to the large *E-Corp* company. A corporation that employs thousands of employees across the globe and operates in various industries.

From the initial meeting with an *E-Corp*'s risk management team and a female CISO Amanda, he knows the following:

- Application's hosting service bills are charged based on its traffic. The budget for the penetration testing session is set to cover up to 300 000 requests.
- Number of accessible pages is not known.
- Amanda requires to receive a penetration testing report detailing the discovered vulnerabilities. It should come as a PDF that follows the official *E-Corp*'s style guide.
- *E-Corp* corporation has an organization profile on the *GitHub* service.
- Target application's URL is `https://e-corp.example.com`

Bob does not know the exact number of pages that should be crawled and scanned, but he can make an educated guess by using the `site:e-corp.example.com` Google dork and observing the number of results returned — roughly a 12 000 pages. Bob sets up a hard limit for *SiteCopier* module to a maximum of 30 000 requests — to cover pages indexed by the Google and to account for possible pages that are not in the Google index.

Also, as he is aware that the corporation employs thousands of people, he considers monitoring possible side-channels for sensitive information. He takes advantage of already implemented tool *PastebinTracker* and sets it up to look for all mentions of the *E-Corp* site URL and the *E-Corp*'s mail suffix. Additionally, he implements a *GitHub* tracking module to continuously monitor additions to the organization's repositories for high-entropy strings.

The testing session is expected to span over the course of five working days and for the whole time, the *PastebinTracker* and the *GitHubTracker* will run. Bob starts these utilities as soon as he starts the testing.

Modules to discover VCS misconfiguration, hidden URL parameters, XSS and the reusable *SQLiFinder* implemented during the previous penetration testing session are also set up to be run.

One final thing that Bob does before launching *ReconJay* utility is implementing new report style named *ECorpFormal*. This format respects style guidelines required by *E-Corp*'s CISO and will be used to generate the final report.

Bob runs the tool and summarizes the results:

- *SiteCopier* crawled 18289 pages.
- Two high-entropy strings were detected by *TokenFinder* module, both of them are false positives — they do not serve as means of authentication.
- Directory listing is disabled on the server.
- No VCS leftover files were detected.
- Parameters `userId`, `ordering` and `category` from `archive/oldportal` subdirectory are vulnerable to SQL injection.
- Parameters `search`, `userName` and `userId` from `prvt/hackaton2012/app` subdirectory are vulnerable to reflected XSS.

All of the mentioned vulnerabilities were generated into the final report which was then converted into requested the PDF format. Additional manual penetration testing that is outside the scope of this case study was conducted during the 5-day long penetration testing session.

During the testing session, the side-channel monitoring tools produced the following results:

- Paste containing user credentials of certain *E-Corp* domain users was detected on the second day of monitoring.
- On the third day of monitoring, 100 pastes containing fake credentials of *E-Corp*'s users were detected.
- On the fourth day of monitoring, a single paste containing mock up HTML design of *E-Corp*'s survey form was detected.
- *E-Corp*'s developers did not push any sensitive information into their public repositories during the whole 5 day monitoring window.

When the credentials dump was detected, Bob immediately contacted the organization's security team and informed them about the leak. They responded promptly by requesting the deletion of a paste in question from *Pastebin.com* and deployed their established mitigation procedure. The mitigation procedure can be observed on the third day of monitoring when 100 pastes containing fake credentials were detected.

Bob compiled all of these results together and presented them at the closing meeting for the *E-Corp*'s risk management team and their CISO. That is where this case study ends.

# Chapter 5

# Testing and Evaluation

This chapter discusses the methodology of testing and evaluation used to verify that the implemented tool operates and performs correctly. The chapter is divided into two main parts — in the first part, the testing process of the implemented tool and its parts is outlined, while in the second part, the results are presented.

## 5.1 Testing & Evaluation Process Design

This section proposes a testing and evaluation process design that is used for evaluating the implemented *ReconJay* tool. Multiple ways of evaluation are used to increase confidence in the implemented tool and to showcase its ability to perform operations it was designed to perform. Following directions of the evaluation process were chosen:

1. **Testing Module Loading Functionality** — Tests and evaluates the tool's key ability to load and execute modules is performed correctly.
2. **Testing Implemented Modules** — Tests all implemented modules, one by one. Evaluates the security scanning functionality provided by the modules.
3. **Testing Standalone PastebinTracker Module** — Similarly evaluates a standalone module that is implemented for monitoring of *Pastebin.com* side channel.
4. **Comparison Testing: Acunetix** — Compares results discovered by the implemented utility to results discovered by the commercial vulnerability scanner.
5. **Testing in Production Environment** — Evaluates results reached by the *ReconJay* tool when run against testing set of production websites that were selected in chapter 2.6.

### Testing Module Loading Functionality

Part of the implemented *ReconJay* tool is a module loading functionality that scans a module source directory for modules and tries to load them. These modules can be dependent one on another and loading and execution need to account for that.

The dependency system needs to be tested on detection of a circular dependency and its capability to recover from states where such dependency is discovered. An occurrence of circular dependency should not prevent the whole tool from running the modules that do not suffer from it.

Following scenarios should produce the following outcomes:

Scenario 1 Independent module is a dependency of one or multiple other modules — no issues during module loading and execution.

Scenario 2 Dependent module (dependency can be satisfied) is a dependency of one or multiple other modules — no issues caused by this dependency during module loading and execution.

Scenario 3 Dependent module (dependency cannot be satisfied) is a dependency of one or more other modules — neither of the modules is executed, they are skipped and an error message is produced.

Scenario 4 Dependent *module 1* is dependent on a dependent *module 2*, while the dependent *module 2* is dependent on *module 1*. Other modules directly or indirectly dependent on module 1 and module 2 exist — none of the modules is executed, the error message is produced.

Outside the module loading functionality, the *ReconJay* tool's ability to run on both *Windows* and *Linux* machine should be verified.

### Testing Implemented Modules

This section presents experiments and scenarios that are designed to evaluate whether implemented modules behave as expected. For each of the modules, a testing scenario or an experiment is presented first, followed by conditions that a module under test must fulfill to pass.

#### Testing TokenFinder Module

In order to test that *TokenFinder* module is capable of detecting secrets in a target application, a test application with hidden secret strings was prepared.

The test application contains randomly generated high-entropy strings of the following properties:

- 5 strings that are 20 characters long and their calculated entropy is below 4.0 (threshold value for being discovered)
- 5 strings that are 20 characters long and their calculated entropy is 4.0 or greater.
- 5 strings that are 19 characters long and their calculated entropy is 4.0 or greater.

For *TokenFinder* module to pass this test, the expected outcome is as follows:

- 5 high-entropy secret strings that are 19 characters long are not detected — minimum required length for a character sequence to be considered a secret string is 20 characters.
- 5 high-entropy secret strings that are 20 characters long and their calculated entropy is 4.0 or greater are detected.
- 5 secret strings with calculated entropy being less than 4.0 are not detected.

### Testing XSSFinder Module

To test the *XSSFinder* module's ability to detect reflected XSS in a target application, a vulnerable testing application was developed.

The test application contains the following parameters with their respective reflection properties:

- 10 parameters that correctly encode user-supplied input and only then reflect it into the page — in both inside and outside HTML attribute contexts.
- 5 parameters that do not perform any input sanitization before reflection.
- 2 hidden parameters that reflect user input in attribute context without sanitizing single and double quotes.
- 2 hidden parameters that reflect user input in attribute context after correctly encoding single and double quotes.
- 2 hidden parameters that reflect user input outside attribute context and fail to sanitize opening character of an HTML tag.
- 2 parameters that reflect user input into the page after converting HTML opening and closing character to a pound character.

For *XSSFinder* module to pass this test, the expected outcome is as follows:

- 10 parameters that correctly encode user supplied input are not reported to be vulnerable to XSS.
- 5 parameters that do not perform any input sanitization are reported as vulnerable to XSS.
- 2 hidden parameters reflected without sanitization in attribute context are reported as vulnerable to XSS, while the 2 hidden parameters that perform correct sanitization are not reported.
- 2 parameters that reflect user input after custom sanitization algorithm is applied are noted as potentially vulnerable to XSS.

### Testing MisconfChecker Module

To evaluate *MisconfChecker* module's ability to discover hidden resources and to detect enabled directory listing functionality, a simple testing scenario is sufficient. The testing application needs to contain commonly known hidden resources and a VCS directory or a file. The server on which the application runs must have the directory listing feature enabled.

For the testing purposes following resources were created in the testing application:

- VCS Resources — `.git` folder, `.git/config` file.
- Other resources — `.env` file, `php.ini` file, `phpMyAdmin` directory.

The module passes the test if it successfully detects these resources and reports the enabled directory listing for a `.git` folder.54

### Testing SiteCopier Module

Unlike other modules, *SiteCopier* module executes no testing that could be classified as unauthorized penetration testing. For this very reason, it is possible to set up a testing scenario that will not need a separate testing application, but instead, will use already existing applications on the Internet.

Main functionality provided by this module is crawling web application and making a local copy of all its important parts that are requested. In order to test whether the module operates correctly, the three applications listed below will be crawled by the module.

- `https://danieldusek.com` — No crawling loops, small web application. Expected number of physical artifacts copied locally: 24
- `http://davidriha.cz` — A crawling loop is present, small web application. Expected number of physical artifacts stored locally: Equal to the requests limitation.
- `https://kentico.com` — No crawling loops, large application. Expected number of physical artifacts copied locally: 3161.

After the crawling is finished, physical artifacts left behind by the implemented tool will be inspected and their match to the crawled applications will be verified.

To pass this test, *SiteCopier* module must crawl the applications successfully and store their responses as its physical artifacts. It must not get stuck in an infinite crawling loop.

### Testing RequestMiner Module

*RequestMiner* module discovers and reports response headers in use, parameters in use and hidden parameters that were possible to detect. In order to test the module's ability to perform these actions, a testing application with the following properties is used:

- Application uses and references these parameters: `page`, `section`, `year`, and `month`.
- Application uses but does not reference parameters: `ban` and `ban_reason`.
- Application uses and does not reference following parameters: `xq24xca` and 2xxxc22.
- Application reports these specific headers outside the standard headers: `Server`, `X-Test-Header` and `Generated-By`.

The module under test should be capable of detecting all headers and parameters used and referenced by the application.

It is admissible for the module not to detect hidden parameters that do not manifest themselves in the application's response content or status code.

Parameters of seemingly random names (`xq24xca`, `2xxxc22`) which are not referenced by the application, and are not among the commonly known parameters, are not expected to be detected.

### Testing Standalone PastebinTracker Module

To evaluate the tool's ability to monitor the *Pastebin.com* side channel, an experiment is to be conducted. Following paragraphs detail a design of this experiment and the expected outcome, that should be produced by the tool under test.

First, a utility capable of posting code snippets to the mentioned service is to be created. The experiment will span over 24 hours, in which this utility will post precisely 200 pastes that will contain pseudo-randomly generated string sequences and a code word. This code word will be known to the *PastebinTracker* utility and will be registered as a monitored keyword.

During the 24 hours window, the utility will monitor *Pastebin.com* for the agreed code word, while the aforementioned code snippet posting utility will post new pastes containing the code word in pseudo-random time intervals. A half (100) of the posted pastes will be removed by the utility after a pseudo-randomly chosen time interval. The deletion request will be delayed for a pseudo-random time from an interval between 10 seconds and 20 minutes after posting.

In order for the *PastebinTracker* to pass this test, all 200 pastes generated by the utility should be recorded.

### Comparison Testing: Acunetix

This evaluation scenario proposes comparing the *ReconJay* tool to the commercial vulnerability scanner *Acunetix*[1]. *Acunetix* scanner was chosen as a reference tool as it is a commonly known vulnerability scanner with a great repertoire of detection capabilities and also because it allows scanning of the prepared *Test PHP Vulnweb*[2] application.

[1] See vendor's website: https://www.acunetix.com/
[2] Available online: http://testphp.vulnweb.com

The mentioned application contains a number of security vulnerabilities that can be detected by both tools. When evaluating results, it must be kept in mind that *Acunetix* performs even destructive and permanent trace leaving interactions with a target. *ReconJay*, on the other hand, sticks to the non-destructive and non-permanent interactions. Due to this, there will be differences in discovered and reported vulnerabilities.

### Testing in Production Environment

In chapter 2, section *Testing Web Application Sets* 2.6, two groups of web applications for testing were selected. The first group is comprised of highly secure applications and the second group is comprised of random applications and applications that are not frequently penetration tested. This chapter contains a testing plan for evaluating the *ReconJay* utility's performance against these targets.

All of the tool's runs will share the same module options (defined in the `options.json` file). Code snippet 5.1 shows used option values.

```
{
    "MisconfChecker": {
        "DELAY": 0.3,
        "RANDOMIZE_SELECTION": "True",
        "MAX_REQUESTS": 500
    },
    "RequestMiner": {
        "DELAY": 0.3,
        "CANARY_LENGTH": 5,
        "MAX_REFLECTION_REQUESTS": 15,
        "URLPARAM_DISCOVERY_HEURISTICS": "START_PAGE",
        "MAX_ACCEPTED_URL_LENGTH": 2000
    },
    "XSSFinder": {
        "DELAY": 0.3
    },
    "SiteCopier": {
        "TOTAL_REQUESTS_LIMITATION": 10000
    },
    "TokenFinder": {
        "ENTROPY_TRESHOLD": 4.0,
        "MIN_TOKEN_LEN": 20
    }
}
```

Code snippet 5.1: Module Options values that are used by the *ReconJay* tool during its evaluation runs.

*ReconJay* will be launched against all of the targets described earlier in this thesis and the reports will be manually reviewed by the author of this thesis. Based on these reports, conclusions will be drawn and the future extensions will be proposed.

The purpose of this testing scenario is not to make a binary decision on whether *ReconJay* passed or failed this test. It is rather to see how the tool performs against the real, existing targets and what vulnerabilities is it capable of detecting.

## 5.2 Testing & Evaluation Results

Actual testing and evaluation results are detailed and discussed in the following paragraphs. This section retains the same structure as the section 5.1 where the testing and evaluation process was proposed.

### Module Loading Functionality Testing Results

This section details the results of the testing scenarios outlined in the section *Testing Module Loading Functionality* 5.1. Table 5.1 presents testing results for given scenarios.

| Scenario no. | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Result | OK | OK | OK | OK |

Table 5.1: Results of module loading scenarios testing (outlined in section 5.1).

All the testing scenarios led to expected outcomes, implemented tool, therefore, passes the module loading functionality tests.

### Implemented Modules Testing Results

This section presents the results of the testing that was conducted according to the testing plan designed in section *Testing Implemented Modules* 5.1. Based on the testing results, conclusions are drawn.

#### TokenFinder Module Testing Results

*TokenFinder* module correctly detected 5 high-entropy secret strings hidden inside the testing application's source code and included them into the generated report. The module also correctly ignored 5 high-entropy secret strings that were only 19 characters long and therefore below the detection threshold. The five secret strings below the calculated entropy threshold were also correctly ignored.

Based on these results, the *TokenFinder* module passes the test.

#### XSSFinder Module Testing Results

*XSSFinder* module provided expected outcome after it was set up to run against the testing application and therefore passes the test. The two parameters which reflect user input after custom sanitization algorithm were reported as `OtherwiseModified` protection level and an advisory paragraph was generated into the final report.

### MisconfChecker Module Testing Results

*MisconfChecker* module discovered the following resources: a `.git` folder, a `.git/config` file, a `.env` file, a `php.ini` file and a `phpMyAdmin` directory.

Enabled directory listing was reported for the `.git` folder. These results correspond to expectations established in the testing plan and the module passes the test.

### SiteCopier Module Testing Results

Results of the test plan execution proposed in section *Testing Implemented Modules* 5.1 are summarized in table 5.2. Discovered number of artifacts matches the expected number of artifacts for each of the testing applications. For each of these applications, the *SiteCopier* module also terminated successfully and did not get stuck in a loop.

| | Expected Artifacts | Discovered Artifacts | Stuck in Loop |
|---|---|---|---|
| danieldusek.com | 24 | 24 | No |
| davidriha.cz | 5000 | 5000 | No |
| kentico.com | 3161 | 3161 | No |

Table 5.2: Table of results reached when testing a *SiteCopier* module.

The second application has a high number of expected artifacts which corresponds to the `TOTAL_REQUESTS_LIMITATION` value (5000). This is due to the fact that the application contains a crawling loop in which it loops until this limitation is reached.

*SiteCopier* module passes the test.

### RequestMiner Module Testing Results

During its test run, *RequestMiner* module discovered the following parameters:

- `page`, `section`, `year`, and `month` — These parameters were discovered because they are explicitly referenced by the testing application and their presence can be detected easily from the `SiteCopier`'s physical artifacts.
- `ban` — This parameter was discovered during the commonly known parameters list testing, due to the altered response from application, when the parameter was added to the URL.

*RequestMiner* failed to discover the following parameters:

- `ban_reason` — This parameter was not detected, because it does not manifest itself by altering the application response.
- `xq24xca`, `2xxxc22` — These two parameters were not discovered, because they are not in the list of commonly known parameters.

All non-standard response headers (`X-Test-Header`, `Server` and `Generated-By`) were discovered. According to criteria defined in section *Testing Implemented Modules* 5.1, the module passes the test.

### Standalone PastebinTracker Module Testing Results

| | | |
|---|---|---|
| Experiment Setup | Snippets posted | 200 |
| | Snippets deleted | 100 |
| Experiment Results | Snippets deleted under 120 seconds since their creation | 12 |
| | Snippets stored by PastebinTracker utility | 188 |

Table 5.3: Interpreted results of the *PastebinTracker* tool experiment designed in section 5.1

After conducting the experiment designed in section 5.1, the results presented by table 5.3 were observed. Experimenting with the *PastebinTracker* tool revealed that the tool is capable of successfully monitoring the *Pastebin.com* side channel for specified code words with one very specific limitation. When the paste containing the code word is removed in under 120 seconds after it is created, it is not detected by the utility.

Further investigation of this behavior uncovered the fact that there is a 2 minutes long caching window between the *Pastebin.com* front-page listing and the caching server that is used by the service's scraping API. This behavior was discussed with the administrators of the service, who claim that it is an intended feature.

In theory, this limitation could be circumvented by scraping the *Pastebin.com* front-page directly and discovering the paste there before it gets deleted. Unfortunately, this would be in a direct breach of the *Terms of Service*[3] document.

Despite the fact that original evaluation criteria required a 100% detection rate, *PastebinTracker* tool passes the test. The discovered limitation of the *Pastebin* service was not known at the time and therefore accounted for.

### Acunetix Comparison Testing Results

Both *ReconJay* tool and *Acunetix* trial version were run against the *Test PHP Vuln Web* application. Table 5.4 compares discovered vulnerabilities that are possible to detect through using non-destructive and non-permanent interactions.

*Acunetix* identifies also other types of vulnerabilities, such as *Server Side Request Forgery*, *Stored XSS*, *Weak Passwords*, or *Brute-forceable Login Forms*, but all of these vulnerabilities require potentially destructive or at least permanent trace leaving interactions. *ReconJay*, on the other hand, only detects vulnerabilities that can be detected without the risk of altering a target application's state. When considering only the vulnerabilities detectable this way, *ReconJay*'s ability to detect them is comparable to that of *Acunetix*.

[3] See: https://pastebin.com/doc_terms_of_service

| | ReconJay | Acunetix |
|---|---|---|
| Reflected XSS | 4 | 4 |
| Enabled Directory Listing | 12 | 12 |
| Backup Files | 1 | 2 |
| `.idea` directory | 1 | 1 |
| Possible sensitive directories | 8 | 3 |

Table 5.4: Comparison table of detected vulnerabilities by *Acunetix* and *ReconJay*.

*ReconJay* was as strong as *Acunetix* at detection of enabled directory listing and slightly better at hidden resource discovery. These results are caused by a better resource payload dictionary that is used by the *ReconJay*. There is one exception to this claim and that is a number of discovered *backup files*. *ReconJay* managed to discover only one backup file of the two present on target application — again, due to different dictionaries in use.

### Testing in Production Environment Results

The two groups of web applications chosen for final evaluation in section 2.6 were tested by the implemented *ReconJay* tool. Table 5.5 details the results of testing the first group of highly secure applications, while table 5.6 shows testing results of the latter group of random web applications that are not frequently tested.

| | XSS | Resources Found | VCS Leftovers | Directory Listing | Detected Parameters | Pages Crawled |
|---|---|---|---|---|---|---|
| flickr.com | 0 | 0 | 0 | 0 | 0 | 10000 |
| bistudio.com | 0 | 0 | 0 | 0 | 0 | 57 |
| bohemia.net | 0 | 0 | 0 | 0 | 10 | 2917 |
| ylands.com | 0 | 0 | 0 | 0 | 0 | 8 |
| ylands.net | 0 | 0 | 0 | 0 | 0 | 8 |
| arma2.com | 0 | 0 | 0 | 0 | 107 | 10000 |
| arma3.com | 0 | 0 | 0 | 0 | 35 | 2874 |
| dayz.com | 0 | 0 | 0 | 0 | 0 | 20 |
| armamobileops.com | 0 | 0 | 0 | 0 | 0 | 41 |
| minidayz.com | 0 | 0 | 0 | 0 | 0 | 12 |
| vigorgame.com | 0 | 0 | 0 | 0 | 0 | 12 |
| hyatt.com | 0 | 0 | 0 | 0 | 2 | N/A |
| world.hyatt.com | 0 | 0 | 0 | 0 | 0 | N/A |
| starbucks.com | 0 | 0 | 0 | 0 | 0 | 6997 |

Table 5.5: Results recorded by *ReconJay* tool when security scanning highly secure web applications (Test Set 1).

Results recorded in table 5.5 confirm the original hypothesis that application from this test set will yield limited to no results.

Several applications also put some obstacles into the path of automated testing. As can be read from the number of issued requests, *ylands* application cannot be efficiently crawled — these applications are almost entirely written in *JavaScript*. The *ReconJay*'s crawler does not support JavaScript-based crawling.

The applications residing on *hyatt.com* and *world.hyatt.com* could not be crawled, as they actively prevent automated crawling and processing of their application – e.g. they refuse connection if they identify it was made using the *python requests* library.

| | XSS | Resources Found | VCS Leftovers | Directory Listing | Detected Parameters | Pages Crawled |
|---|---|---|---|---|---|---|
| davidriha.cz | 3 | 1 | 1 | 2 | 8 | 10000 |
| danieldusek.com | 0 | 0 | 0 | 0 | 0 | 57 |
| netsearch.cz | 0 | 0 | 0 | 0 | 25 | 138 |
| nesad.fit.vutbr.cz | 0 | 0 | 0 | 0 | 1 | 38 |
| testphp.vulnweb.com | 4 | 12 | 1 | 10 | 7 | 57 |

Table 5.6: Results recorded by *ReconJay* tool when security scanning a selection of applications that are not frequently penetration tested (Test Set 2).

Testing of applications from the *Test Set 2* showed that *ReconJay* tool is capable of detecting vulnerabilities in real applications that are not under frequent penetration testing. For two applications from this test set, places vulnerable to reflected XSS were discovered.

When testing *davidriha.cz* application, it was also discovered that a remote host where the application physically resides, is capable of detecting that it is being scanned for vulnerabilities. Some of the requests on hidden resources were automatically sinked to non-existing `www.ihateexploits.cc` domain.

## 5.3 Future Extensions

This section presents a few extensions that are planned to be implemented in the future. Some of these extensions are a direct result of evaluation and testing phase, while other proposed extensions are completely standalone improvements of existing functionality. They focus mainly on increasing the scope of vulnerability testing that would lead to a more broad repertoire of detected vulnerabilities and vulnerable patterns.

### PastebinTracker Detection Improvements

The evaluation phase of the standalone *PastebinTracker* module discovered that when a paste was deleted under 120 seconds since its creation, the module was unable to detect it. Proposed future extension addresses these issues and manages to detect newly created pastes directly from the *Pastebin.com*'s front page.

Significant challenges in implementing this extension exist. One of these challenges is that implementing and using this functionality is a violation of the *Pastebin*'s *Terms of Service* document. The other challenge lies in implemented protection against this form of scraping on the *Pastebin*'s side — a way to avoid getting detected by the service's anti-scraping functionality needs to be designed.

### More Vulnerability and Information Disclosure Detection Modules

The *ReconJay* utility was designed to be easily extended by new modules. Such modules can extend and build upon already existing modules or implement completely new security scanning and evaluation functionality.

One of such modules was already hinted in the first hypothetical case study 4.7 — *SQLiFinder* module that is capable of detecting error-based SQL Injection type of vulnerability. Other possible extensions are standalone modules for monitoring side-channels such as *GitHub*, *GitLab* and other existing public boards for sharing information.

### Improving RequestMiner Module Functionality

One of the core modules, *RequestMiner* is dedicated to acquiring information about target application based on the physical artifacts collected by the *SiteCopier* module and on its interactions with a target application. Extending this module's functionality, e.g. by the ability to brute-force some of the existing but hidden query string parameters, could bring a benefit to a penetration tester using this tool.

Another possible extension would be to implement functionality for observing a target application's behavior when provided with various request headers. The benefit here lies in a possibility of discovering reflected request header values, or in opening doors to debug-only functionality, not intended for the public.

# Chapter 6

# Conclusion

This thesis had two goals — firstly, to propose a generally applicable approach to web application penetration testing that utilizes only non-destructive and non-permanent interactions with a target application, and secondly, to implement a tool that would follow the proposed approach and could be used for penetration testing. Both of these goals were achieved — the approach is proposed and subsequently the *ReconJay* tool following it is designed and implemented.

There are 5 items in the master thesis specifications that place certain requirements on the final outcome of this work. The first item requires studying approaches, methods and existing software used for web application penetration testing. This work studies the two most commonly known and used methodologies for penetration testing, five non-invasive penetration testing techniques and eight penetration testing tools.

Another requirement, specified by the second item, is to design the aforementioned approach. Author of this thesis proposes and designs the approach comprised of 3 significant steps. The steps cover typical stages of the penetration testing methodologies that were studied, but utilizes only non-destructive and non-permanent interactions.

The third item deals with the implementation of a tool that follows the designed approach. It was addressed by *ReconJay* tool being developed. This tool can be used for non-destructive penetration testing and follows the principles outlined in the approach. It supports the detection of reflected XSS vulnerabilities, reflected query string parameters, certain server misconfigurations and hidden resource enumeration. Except for detecting the vulnerable patterns, the tool is also capable of providing security relevant information about a target application to a tester. Example of this is listing the existing query string parameters, non-standard headers or leftover version control files. Main original contribution of this work is an implementation of the standalone *PastebinTracker* module that serves for monitoring the *Pastebin.com* side-channel for information leakage. This module was developed as a supportive tool to be used together with the *ReconJay*. Finally, two case studies showcasing the tools were presented. One of them deals with penetration of a small web application, while the other one focuses on a large corporation target.

The fourth item requires an evaluation of the implemented tool. Testing and evaluation is approached from multiple perspectives. First, the tool's capability to load its modules is tested by executing corner-case scenarios. This is followed by testing of each of the five implemented modules on their capability to perform what they were built for. The results

are presented and provided outputs are explained. After that, an experiment with the standalone module *PastebinTracker* is conducted. The experiment results testify that it is able to operate correctly and potentially discover leaking information. The tool's limitation to detect information that is deleted within a 120 seconds time window after its creation is revealed, then closely explained and reasoned about. A comparison of the *ReconJay* tool to a commercial *Acunetix* vulnerability scanner follows next. The comparison testing revealed that in non-destructive and non-permanent penetration testing, the *ReconJay* tool outperforms the *Acunetix* scanner in identification of potentially vulnerable resources. *ReconJay* detected 8 occurrences of potentially vulnerable resources, while *Acunetix* was able to detect only 3 of them. *ReconJay* performs just as well as *Acunetix* in reflected XSS and enabled directory listing detection. And finally, the tool's performance is tested on testing application sets designed whilst working on item one. The tool did not detect any vulnerabilities in expected highly secure applications. It did manage to discover XSS vulnerabilities, enabled directory listing and hidden resources disclosure in 2 of the 5 applications in the set of applications that are not frequently penetration tested.

Item five of the specification requires a user documentation to be made. The user documentation is delivered in appendix B and contains installation instructions and examples of a proper tool usage. Item five also requires an outlining of the future extensions of the tool and explaining their benefits. This was mostly covered as a followup to the evaluation and testing chapter and proposed extensions include: extending the existing functionality of the *RequestMiner* module, for example by adding support for brute-forcing query string parameters or request headers, and also implementing new modules for monitoring side-channels other than *Pastebin.com*.

An extension that could be made in the nearest future would focus on improving the *RequestMiner*'s functionality. The addition of brute-force guessing of query string parameter names and hidden request header names would further improve the tool's reflected XSS detection capabilities.

The extension plan for the far future encompasses implementing the *SQLiFinder* module with advanced capabilities of SQL Injection detection. Such extension must still adhere to the non-destructive and non-permanent requirements and needs to come up with a smart way to detect existing SQL injection in a target application.

This work was presented at conference *Excel@FIT2019*.

# Glossary

**Blue Team** is a team of security people, typically inside a company. The blue team members are actively trying to prevent security incidents from happening and to improve the overall security of the company. 16

**BurpSuite** is a penetration testing tool with graphical interface and a various tools, both automated and manual that make penetration testing of web applications easier. 7, 8, 19

**CAL9000** is a software comprised of several other software tools that ease the manual penetration testing. The project was allegedly abandoned by its project team. 7

**Internet Archive** is a non-profit organization that is building a digital library of the internet. Snapshot of various web sites and their pages are taken in different points in time. Users can request page snapshot being taken and can also browse these snapshots for any of the recorded web pages [2]. 17

**JSON** The abbreviation itself stands for *Javascript Object Notation* and is the way that objects are standardly described in JavaScript. These days it is used by a wide variety of software, because it allows easy exchange of structured data. 16, 17

**Netsparker** is an automated web application security scanner that scans the web application for a number of known vulnerabilities from various vulnerability classes. The tool provides fairly accurate results with working proof of concepts that demonstrate discovered vulnerabilities. 8

**Secret** is a password or a passphrase that should not be disclosed to the third parties and people outside the need-to-know basis. Example of secrets can be access tokens for user accounts or production applications' environments. 16

**WebScarab** is a software designed to serve as a proxy between the user browser and web server that allows the penetration tester to alter the requests sent to and from the webserver. 7

**ZAP** The abbreviation stands for *Zed Attack Proxy* and it is a tool helping with both manual and automated penetration testing of web applications. Maintained and improved by the open-source community. 7

# Bibliography


[1] Trends in Consumer Mobility Report. Technical Report 1.415.913.4416. Bank of America. August 2017.

[2] archive.org: About the Internet Archive. 2018, (accessed January 1, 2019). Retrieved from: https://archive.org/about/

[3] Barber, R.: Hackers profiled—who are they and what are their motivations? *Computer Fraud & Security*. vol. 2001, no. 2. 2001: pp. 14–17. ISSN 1361-3723. doi:10.1016/S1361-3723(01)02017-6.

[4] Beazley, D.; Jones, B. K.: *Python Cookbook: Recipes for Mastering Python 3*. „ O'Reilly Media, Inc.". 2013. ISBN 978-1-449-34037-7.

[5] Chess, B.; West, J.: *Secure programming with static analysis*. Pearson Education. 2007. ISBN 0-321-42477-8.

[6] Cimpanu, C.: Gamarue Botnet Uses Hijacked WordPress Sites to Send Spam with JS Payloads. April 2016, (accessed November 21, 2018). Retrieved from: https://news.softpedia.com/news/gamarue-botnet-uses-hijacked-wordpress-sites-to-send-spam-with-js-payloads-502842.shtml

[7] Cormode, G.; Krishnamurthy, B.: Key differences between Web 1.0 and Web 2.0. *First Monday*. vol. 13, no. 6. 2008. ISSN 13960466. doi:10.5210/fm.v13i6.2125.

[8] Engebretson, P.: *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Syngress Publishing. second edition. 2013. ISBN 9780124116412.

[9] Ernst, M. D.: Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR. 2003. ISBN 0-7695-1877-X. ISSN 0270-5257. pp. 24–27. doi:10.1109/ICSE.2003.1201290.

[10] Fielding, R.; Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. RFC Editor. June 2014.

[11] Foudil, E.: A Method for Web Security Policies. Informational securitytxt-rfc. ietf.org. January 2019.

[12] Garsiel, T.; Irish, P.: How browsers work: Behind the scenes of modern web browsers. *Google Project, August*. 2011.

[13] Google: Refine web searches. 2019 (accessed January 1, 2019).
Retrieved from: https://support.google.com/websearch/answer/2466433

[14] Hatmaker, T.: Facebook password stealing software comes packed with a trojan that steals your passwords. August 2017, (accessed November 17, 2018).
Retrieved from:
https://techcrunch.com/2017/08/10/facebook-password-stealing-software-comes-packed-with-a-trojan-that-steals-your-passwords/

[15] Hudák, P.: *Analysis of DNS in cybersecurity*. Master thesis. Masaryk University, Faculty of Informatics, Brno. 2017.

[16] Jadhav, M. A.; Sawant, B. R.; Deshmukh, A.: Single page application using angularjs. *International Journal of Computer Science and Information Technologies*. vol. 6, no. 3. 2015: pp. 2876–2879. ISSN 0975-9646.

[17] Karpowicz, R.: Custom file formats. Jun 7, 2017 (accessed January 1, 2019).
Retrieved from:
https://github.com/auth0/repo-supervisor/wiki/Custom-file-formats

[18] Kaspersky: What is a Black-Hat hacker? 2016, (accessed November 17, 2018).
Retrieved from:
https://www.kaspersky.com/resource-center/threats/black-hat-hacker

[19] Kettle, J.: So you want to be a web security researcher? May 2018, (accessed November 17, 2018).
Retrieved from: https://portswigger.net/blog/so-you-want-to-be-a-web-security-researcher

[20] Koster, M.: A Method for Web Robots Control. Informational norobots-rfc. robotstxt.org. December 1996. http://www.robotstxt.org/norobots-rfc.txt.
Retrieved from: http://www.robotstxt.org/norobots-rfc.txt

[21] Langston, M.: Six Best Practices for Securing a Robust Domain Name System (DNS) Infrastructure. Feb 6, 2017 (accessed January 27, 2019).
Retrieved from:
https://insights.sei.cmu.edu/sei_blog/2017/02/six-best-practices-for-securing-a-robust-domain-name-system-dns-infrastructure.html

[22] Martin, R. C.: *Clean code: a handbook of agile software craftsmanship*. Pearson Education. 2009. ISBN 9780136083238.

[23] McAfee: 7 Types of Hacker Motivations. March 2011, (accessed November 17, 2018).
Retrieved from: https://securingtomorrow.mcafee.com/consumer/family-safety/7-types-of-hacker-motivations/

[24] McNally, C.: maK-/parameth. 2018, (accessed January 1, 2019).
Retrieved from: https://github.com/maK-/parameth/

[25] Mockapetris, P.: Domain names - implementation and specification. STD 13. RFC Editor. November 1987.

[26] Mohamed, R.: Assessment of Web Scanner Tools. *International Journal of Computer Applications (0975 – 8887)*. vol. 13, no. 6. January 2016. ISSN 0975-8887. doi:10.1.1.735.7781.

[27] Muller, A.; Meucci, M.; Keary, E.; et al.: OWASP testing guide 4.0. 2014 (accessed November 21, 2018).

[28] Nottingham, M.; Hammer-Lahav, E.: Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785. RFC Editor. April 2010.

[29] OWASP: About The Open Web Application Security Project. September 2018, (accessed November 25, 2018).
Retrieved from: https://www.owasp.org/index.php/About_The_Open_Web_Application_Security_Project

[30] Rouse, M.: What is white hat? January 2018, (accessed November 17, 2018).
Retrieved from: https://searchsecurity.techtarget.com/definition/white-hat

[31] Rouse, M.: What is a script kiddy (or script kiddie). June 2007, (accessed November 17, 2018).
Retrieved from:
https://searchmidmarketsecurity.techtarget.com/definition/script-kiddy

[32] Rouse, M.: What is black hat? June 2017, (accessed November 17, 2018).
Retrieved from: https://searchsecurity.techtarget.com/definition/black-hat

[33] Samir Patel, R. A.: Directory Browse <directoryBrowse>. Sep 26, 2016 (accessed February 9. 2019).
Retrieved from: https://docs.microsoft.com/en-us/iis/configuration/system.webserver/directorybrowse

[34] Shannon, C. E.: A mathematical theory of communication. *Bell system technical journal*. vol. 27, no. 3. 1948: pp. 379–423.

[35] Shirokova, A.; Valeros, V.: An Overview of the WCMS Brute-forcing Malware Landscape. *The Journal on Cybercrime & Digital Investigations*. vol. 3, no. 1. 2017: pp. 20–29. ISSN 2494-2715. doi:10.18464/cybin.v3i1.18.

[36] Simmonds, A.; Sandilands, P.; Van Ekert, L.: An ontology for network security attacks. In *Asian Applied Computing Conference*. Springer. 2004. ISBN 978-3-540-23659-7. pp. 317–323. doi:10.1007/978-3-540-30176-9_41.

[37] Valeros, V.: Make It Count: an Analysis of a Brute-forcing Botnet. *The Journal on Cybercrime & Digital Investigations*. vol. 1, no. 1. 2016. ISSN 2494-2715. doi:10.18464/cybin.v1i1.5.

[38] Van Hattem, R.: *Mastering Python*. Packt Publishing Ltd. 2016. ISBN 978-1-78528-972-9.

[39] Wichmann, B.; Canning, A.; Clutterbuck, D.; et al.: Industrial perspective on static analysis. *Software Engineering Journal*. vol. 10, no. 2. 1995: pp. 69–75. ISSN 0268-6961. doi:10.1049/sej.1995.0010.

[40] Zalewski, M.: *The Tangled Web: A Guide to Securing Modern Web Applications*. San Francisco, CA, USA: No Starch Press. first edition. 2011. ISBN 1593273886, 9781593273880.

# Appendices

# Appendix A

# Presentation styles

The appendix A demonstrates the two presentation styles available in the implemented *ReconJay* utility:

1. *BWFormal* — Black & White, Formal A.1
2. *Plaintext* — Basic plaintext style A.2

# Web Application Penetration Testing Report

*Generated by ReconJay Tool*

Target: http://davidriha.cz

## XSSFinder

An *XSSFinder* module takes advantage of information gathered by both *SiteCopier* and *RequestMiner* modules. It looks for content supplied by the user that is reflected back into the page and then determines whether the website author implemented sufficient protection against XSS (typically encoding).

XSSFinder is aware of the context in which the payload is reflected and before reporting discovered XSS, it verifies that necessary preconditions were satisfied for the finding, thus avoiding false positives.

Scan detected **3** reflected XSS vulnerabilities.

| URL | Vulnerable parameter | Protection | Rendering Context |
|---|---|---|---|
| http://davidriha.cz/?month=4&page=zapasy&sekce=kalendar&year=2019 | year | None | — |
| http://davidriha.cz/?month=4&page=zapasy&sekce=kalendar&year=2019 | year | None | — |
| http://davidriha.cz/?month=4&page=zapasy&sekce=kalendar&year=2019 | year | None | — |

## MisconfChecker

*MisconfChecker* module checks for configuration errors in deployed application, such as enabled directory listing, VCS structures put into the production and hidden resources that are available, but not meant to be seen.

Other interesting **resources** were found in the following locations:

- http://davidriha.cz/lightbox

Figure A.1: Presentation style: Black and White, Formal

```
DOCUMENT_NAME: Vulnerability Report (2019-05-03_140648_LCDHK)

+=======================================================
|
| WEB APPLICATION PENETRATION TESTING REPORT
| Generated by: ReconJay tool
| Target: http://davidriha.cz
|
+=======================================================


|================================================|
|-> Module: XSSFinder
|================================================|

An XSSFinder module takes advantage of information gathered by both SiteCopier and
RequestMiner modules. It looks for content supplied by the user that is reflected
back into the page and then determines whether the website author implemented
sufficient protection against XSS (typically encoding).

Scan detected 3 reflected XSS vulnerabilities.Format: URL | Vulnerable parameter name | Protection | Rendering

http://davidriha.cz/?month=5&page=zapasy&sekce=kalendar&year=2019|year|None|-http://davidriha.cz/?month=5&page=
month=5&page=zapasy&sekce=kalendar&year=2019|year|None|-

|================================================|
|-> Module: MisconfChecker
|================================================|


MisconfChecker module checks for configuration errors in deployed application, such as enabled directory
listing, VCS structures put into the production and hidden resources that are available, but not meant to
be seen.


MisconfChecker did not collect any presentable data.
```

Figure A.2: Presentation style: Plaintext, default fall-back when style name is not recognized, or not supplied.

# Appendix B

# ReconJay Tool — User Manual

The appendix B contains a user manual for the implemented *ReconJay* tool. Examples of the proper tool usage are included.

## Installation

On the optical disk attached to the thesis, there is a *ReconJay* folder which contains source files of the implemented tool. Among these files, `requirements.txt` file is located. This file specifies PIP packages that need to be downloaded and installed prior to the *ReconJay*'s run. They can be installed by running one of the commands presented in code snippet B.1.

```
pip install -r requirements.txt
OR
python -m pip install -r requirements.txt
```

Code snippet B.1: Installation of PIP dependencies.

For a problem-free execution of implemented tools, Python version 3.6 or higher is recommended.

## Usage

Two executable *Python* scripts are attached to this work. First of them is the `ReconJay.py` that accepts only a single parameter — the URL address of a web application. After the command presented in code snippet B.2 is run, it will execute non-destructive penetration testing of an application that resides on provided URL.

```
python ReconJay.py http://example.com
```

Code snippet B.2: *ReconJay* tool: Example of use.

Additionally, it is possible to set certain options for each module loaded by the *ReconJay* tool, on per module basis. For this purpose, `options.json` file can be used. Inside this file, a dictionary object is present, where its keys correspond to module names. Under

these keys, certain module settings are configurable. A complete demonstration of these settings and possible values is shown in the `options.json` file inside `ReconJay` folder on the attached optical disk.

The second executable *Python* script, `PastebinTracker.py`, is intended for continuous monitoring of the *Pastebin.com* site for interesting pastes that appear there.

A tester must provide the script with keywords for which to look in newly posted pastes. This is realized through `-k` or `--keywords-file` parameter. Provided file should contain the list of keywords, each of them on a new line. When no keywords file is specified, `keywords.txt` file is used.

They can also provide `--fetch-interval` value to specify how often will the script retrieve information about new pastes — a number of seconds is expected. Fetch timeout defaults to 10 seconds when not set.

Last parameter that a tester can specify is an output directory to which the interesting pastes should be stored. This parameter defaults to the `recorded_pastes` value and can be adjusted by `-o` or `--output-dir` parameter.

Code snippet B.3 shows examples of how the script can be run.

```
python PastebinTracker.py --output-dir pastes
python PastebinTracker.py --fetch-timeout 5 -k keywords_creditcards.txt \
                          -o cards
```

Code snippet B.3: *PastebinTracker* tool: Example of use.

# Appendix C

# Contents of the Optical Disc

On the optical disc attached to this thesis, following contents can be found:

- `ReconJay/` containing complete source codes of the implemented tool.
- `thesis/` containing electronic version of this document in `.pdf`.
- `thesis-latex/` containing source files of this thesis.
- `report-samples/` containing sample reports produced by the *ReconJay* application.
- `readme.txt` containing brief description of the files on the optical disc and a user manual for the *ReconJay* applicaiton.