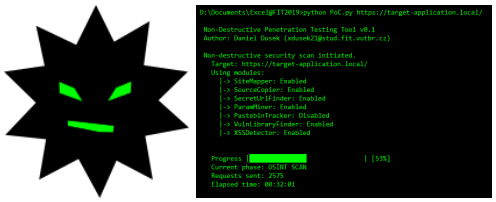
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Non-destructive Security Testing for Web Apps

Daniel Dušek*



Abstract

Penetration testing of a company's web application on a regular basis comes either with the fear of possible downtime, or with the need for setting up the test environment. This work aims to offer the solution that will both minimize the fear and remove the need for a separate testing environment and will also enable automated regular security scans. This paper presents a new, generally applicable approach to the web application penetration testing that leans heavily towards the automation while at the same time is non-destructive and leaves no permanent traces of the testing on the target application. The proposed approach can serve as a guiding line for implementation of a tool that automates a portion of the standard penetration testing and eases a tester's work. Applicability of this approach is supported through the proof of concept implementation that follows it. A reader of the paper can adapt the proposed approach to developing their own penetration testing tool.

Keywords: Penetration Testing Automation — Web Application Penetration Testing — Web Application Penetration Testing Methodology — Open Source Intelligence Automation — Automated Security Testing

Supplementary Material: *N/A*

*xdusek21@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

One of today's most interesting areas for penetration testing is the realm of web applications. Every company these days has its own website and a lot of these companies even offer some sort of the service through web. Penetration testing of such applications is usually desired by some of the companies, but it does not come without a cost.

Conducting penetration testing on production systems has the potential for causing downtime of the service, which is something that most companies prefer to avoid. One possible solution to mitigate the risk of downtime is to deploy a separate copy of the production environment dedicated for testing purposes.

In this paper, I would like to propose a different solution — an alternative approach to web application penetration testing. The approach that heavily leans towards the automation while utilizing only non-destructive and non-permanent interactions with the target applications. This way, the tools implementing the approach could be run with a high degree of confidence against the production environments without significant risks of the downtime.

The proposed use of the automation aims to automate most of the manual and repetitive tasks that would normally occupy a penetration tester. A tester can then spend more time by testing complex scenarios that are hard or impossible to automate. Additionally, a delegation of testing tasks to the computer increases the amount of information that can be processed and

evaluated in the course of the testing session — an amount that would not be otherwise humanly possible.

The core part of this paper is the proposal of the generally applicable approach to web application penetration testing that supports non-destructive and no permanent trace leaving testing of the target application. Proposed approach should lean heavily towards the automation and should serve as a guiding line for implementing tools for the described type of penetration testing. As a bonus part of the paper, a tool implementing principles of the proposed approach is delivered. This demonstrates the possibility of turning the approach into the working testing tool.

My solution develops an approach to web application penetration testing that reduces the amount of manual work for a penetration tester and enables them to write their own automated penetration testing tools that can be run against production environments.

My developed approach has been tested and supported by implementing the automated tool for penetration testing purposes, which follows the proposed principles. Work on the tool is still in progress and will be further extended in the future.

2. Web Application Penetration Testing

A lot has changed on the Internet in the past few years and the focus of classic web applications has shifted as well. It is more about communication and cooperation than about passive document sharing and reading. This shift towards more active user engagement and contribution is known as a transition from *Web 1.0* to *Web 2.0* [1] and it drastically changes the way web applications are built and used. Social networks like Facebook or Twitter, team communication tools like Slack or MS Teams and even web email clients are focused on supporting user cooperation and contribution too. Online and mobile banking applications are also becoming the norm [2].

In order to power *Web 2.0*, loads of new languages and technology means were invented. JavaScript, for example, is way more common then it used to be and the new libraries and frameworks are released quite frequently. Modern ways of developing web application — like for example using *Node.js* — come out of the box with an available package manager that allows developers to reuse other developer's code in their application very easily. Another very common pattern that can be spotted is web applications referencing sources from different applications located on the Internet (e.g. *Google Analytics Javascript*, a code for tracking user activity on the web).

While it is true that this all heavily improves the user experience for both the developers and the end users, with all the new technology being introduced this quickly, the attack surface of these systems increases too. In combination with the ongoing competition between browser vendors for new HTML5, CSS3 and JavaScript features [3], which has led to vulnerabilities in the past [3], web application security becomes a very current and broad topic.

At the time of writing, most of the internet facing web applications is likely impacted by some form of penetration testing — either from well-intended or malicious reasons. The need to secure web applications against these attacks is generally acknowledged.

Actors that conduct penetration testing of web applications are often classified into the three categories [4, 5]:

1. Researchers (*White hats*),
2. malicious hackers (*Black hats*),
3. script kiddies.

Any of the aforementioned actors can also deploy *automated or autonomous systems* that crawl the internet-facing web applications and scan them for vulnerabilities.

An example of **automated systems** are commercial, general purpose vulnerability scanners such as *Netsparker*[1] or *BurpSuite*[2]. Penetration testers deploy these tools against their target and launch the scan. Once the scan finishes, they review vulnerability report generated by the automated system and take appropriate steps that reflect their agendas.

Certain systems deployed on the Internet are not only automated but also **autonomous**. Example of such a system is the *Andromeda/Gamarue* botnet that used to search for *Wordpress* instances on the Internet and tried to break into their administrations [6]. After breaking into the administration, it used its control over the *Wordpress* instance to spread spam [7].

In the field of penetration testing, the two approaches stand out. It is the *Zero Entry Hacking* [8] or *OWASP Web Application Security Testing* [9].

ZEH approach describes penetration testing methodology that starts with a large amount of broad information that is processed in the course of penetration testing. It consists of 4 to 5 (opinions differ) phases where each next phase processes smaller amount of more concrete information. This leads to the final output — the low amount of highly valuable information about the security state of the target application [8].

[1] Vendor's website: https://www.netsparker.com/
[2] Vendor's website: https://portswigger.net/burp

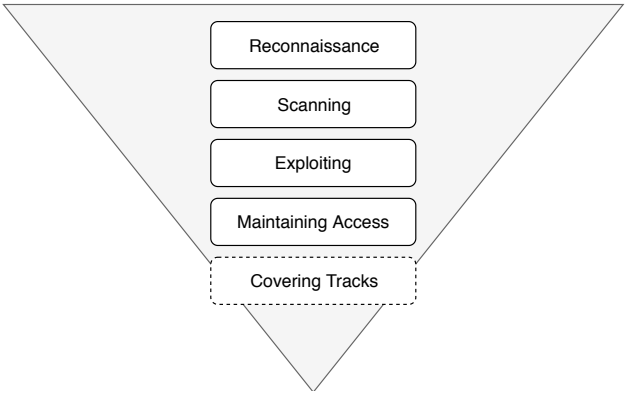


Figure 1. The Zero Entry Hacking Penetration Testing methodology visualized as a reverse triangle diagram, illustrating the decreasing scope and amount of information when progressing from the initial phases to the final phases [8]. The diagram is extended by the fifth phase dedicated to covering the tracks of successful penetration.

Four to five phases are often visualized as an inverse triangle, as seen in Figure 1.

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 [9]. 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 [10]. Differently from the previously mentioned methodology, the OTG proposes a different approach to discovering vulnerabilities — it separates the testing into two phases, passive and active. In the passive phase, a tester is supposed to interact and play around with a target application and passively collect information about the application in the process. In the active phase, a tester is expected to start actively testing for 11 OWASP defined subsections by using their respective methodologies.

Destructiveness of Penetration Testing

While *ZEH* and *OTG* provide guidelines and recommendations on how to conduct penetration testing of a target system, they do not really focus on the means of the testing itself. It is left up to a penetration tester to analyze what tools and testing techniques are appropriate to deploy.

Should a tester be tasked with testing a live environment used by other users, the possibility of damaging it during the testing must be considered beforehand. **In context of this paper, a destructive penetration testing is any form of penetration testing that will result in altering the state of a target application**.

When we consider software like *BurpSuite* or *Netsparker* and look at how their vulnerability scanning/intrusion functionality works, we will see that they can not be safely run without disabling some of their features first.

Basic operating mode of such scanners involves sending huge volume of potentially destructive HTTP requests (such are for example requests made using the `POST`, `PUT` or `DELETE` verb [11]). These requests are loaded with vulnerability-specific payloads and the scanners inspect how the application responds to them in hopes of discovering its weak spots [12, 13]. Running the tools without considering this behavior introduces risks to the target application.

Risks of Destructive Testing

When destructive penetration testing is conducted, a target web application faces some risks. These risks may include:

1. **Downtime** — application becomes unavailable for all the users.
2. **Slow response time** — application is available, but its performance degrades, resulting in delayed reply to its users.
3. **User annoyance** — application starts malfunctioning as a result of a successful penetration by the scanner and its users notice it.
4. **Damage to application or user data**.
5. **Damage to reputation**.

3. Proposed Approach to Web Application Penetration Testing

In this section, the definition of non-destructive and non-permanent actions is presented, as understood in the context of this article. After that, the proposed approach to web application penetration testing and its automation is presented. Three concrete scan phases — *OSINT Scan Phase, 3rd Party Tools Phase* and *OSINT Report* phase are closely described.

This section is the core part of the paper and the proposed approach is the main contribution.

Non-destructive and Non-permanent Actions

A typical communication between a web application and a server occurs over the HTTP using the *HTTP requests* [11].

In this paper *non-destructive* and *non-permanent* interactions are considered to be either the *HTTP requests* sent by so called *Safe Methods*, or subset of parametrized and non-parametrized methods of so called *Idempotent Methods* group [11].

Safe methods are described as essentially read-only methods which does not change the state of data on the

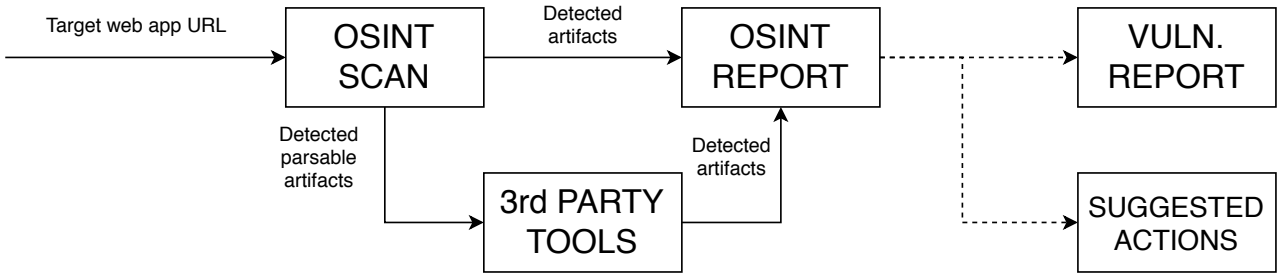


Figure 2. Block schema of the high-level approach proposal. Initially, only the target web application URL is on the input and as the first step, the *OSINT Scan phase* is initiated. Based on the type of detected artifacts (output) the *3rd Party Tools phase* may be initiated to process more complex results. All the detected results that do not require further processing are fed as an input of the *OSINT Report phase* where final output for a tester is generated. Given by the specific requirements on the testing session, various types of reports may be produced. In the figure, a *Vulnerability Report* and *Suggested Actions* reports are used as an example.

remote server. Requests with `GET` and `HEAD` method falls into this category [11]. Requests with `OPTIONS` and `TRACE` methods should have no side effects [11]. These methods are considered to be *non-permanent* and *non-destructive* in the context of this paper.

Of course, it is possible for any of the considered *non-permanent* and *non-destructive* interactions to be destructive when executed against the target application that is implemented in a way that contradicts to HTTP specification. Should this be the case, the RFC suggests that the resource owner is responsible [11].

Any HTTP request that utilizes `HEAD`, `GET`, `OPTIONS` or `TRACE` verb is considered to be non-destructive and non-permanent trace leaving request in the context of this paper. Furthermore, `POST` requests with the subjectively high confidence in being non-destructive and non-permanent trace leaving requests are admissible (e.g. `POST` requests targeting the search service).

High-level Approach Proposal

Approach that I am proposing expects URL address to be initially the only input provided to a penetration tester. In the very first phase (*OSINT Scan*; see Figure 2), a tester should collect as much openly available information as possible about the application residing on provided URL. Collected information should be then filtered, processed (*3rd Party Tools* and *OSINT Report* phases; see Figure 2) and categorized. Separate categories are use-case specific and it is up to a penetration tester to properly establish them.

Based on the category to which collected information belongs, additional category-specific processing steps, should be taken (both *3rd Party Tools* and *OSINT Report* phases; see Figure 2).

Finally, a report summarizing the findings will be generated from the results of all the steps taken and provided back to a penetration tester. Depending on the nature of information collected and processed, the output report can be presented as a list of the discovered vulnerabilities, suggested actions or something else entirely.

OSINT Scan Phase — This phase represents the first and the most complex phase of the proposed approach. A tester specifies their areas of interest and requirements, researches and implements tools to scan these areas in an automated manner and runs the scan. This phase is proposed to be split into the following sub-steps:

1. **Define** — Decision what areas of the target application are to be scanned, and how the results are going to be presented.
2. **Research** — Research what tools can be used to fulfill the requirements and to cover the areas specified in the previous step.
3. **Implement** — Implementation and chaining of the tools selected in the previous step. If tester implements their own tool or wrapper around the 3rd party tool, re-usability and automation should be kept in mind.
4. **Run** — Execution of the automated penetration testing. Run output (the artifacts) is passed as an input to the *3rd Party Tools* or *OSINT Report* phase of the approach.

After this phase is completed high volume of available open-source intelligence should be collected and classified into *non-parsable*, resp. *parsable* artifacts, which are fed as an input into *OSINT Report*, resp. *3rd Party Tools* phases.

An example of non-parsable artifacts that are fed into the *OSINT Report* phase is a list of missing security headers of the target page. An example of parsable artifacts fed into the *3rd Party Tools* phase for further processing is a list of discovered query string parame-

ters that are candidates to be attacked by reflected XSS detector.

3rd Party Tools Phase — Designed to use of already existing and developed tools for further processing of data or penetration testing tasks. This phase is tightly connected to the previous phase as in the previous phase, wrappers for 3rd party tools are implemented.

Note that this phase does not prescribe the use of the only and exclusively 3rd party tools. A tester can use their previously implemented utilities that specialize in specific tasks benefiting the current testing session. In fact, the re-use of already implemented utilities is highly recommended.

Given the example of a list of query strings to be attacked as an input into this phase, the output of this phase takes form of a list of query string parameters vulnerable to the reflected XSS attack.

OSINT Report Phase — Dedicated to working with the acquired artifacts that are not further parsable. The artifacts are acquired from both the *OSINT Scan* and *3rd Party Tool* phases and in this phase are compiled into the final report (the form of this report has been decided in the *OSINT Scan phase, Define step*.

An output of this phase should be affected by the requirements on the penetration testing as specified by the party that has requested it. In general, the output could take the form of discovered vulnerabilities report, the list of sensitive information or the information and pentesting pathways worth exploring.

To illustrate possible output artifacts that are expected to be processed in the final *OSINT Report* phase: application secrets or access tokens, vulnerable libraries in use, leaking personal information, system users with weak or well-known passwords.

4. Implementation of the Penetration Testing Tool

Goal of this section is to demonstrate that it is possible to implement the tool that follows the approach proposed in section 3. The proposed approach aims to be generally applicable, but the implementation does not. The implementation aims to be one possible take on web application penetration testing automation.

This section is a bonus part of the paper and is meant to showcase the tool that is currently being implemented and happens to reflect aforementioned principles.

The implemented solution takes advantage of the python's handy packaging system and its cross-platform capabilities. The tool does not offer any GUI functionality and is intended to be run as a command line utility.

At the time of writing, the implemented tool includes functionality that covers discovering and testing for the following scenarios:

1. Discovering URL parameters,
2. discovering *IIS* and *VCS* miss-configurations,
3. discovering not linked (possibly secret) URLs,
4. real-time monitoring of side channels,
5. listing secrets and access tokens,
6. evaluating data transmission security,
7. site map generation.

The implemented tool is capable of discovering the following vulnerabilities and weak patterns:

1. Reflected XSS,
2. enabled directory browsing,
3. missing subresource integrity checks,
4. use of vulnerable JavaScript libraries.

To respect *non-destructive* and *non-permanent* interaction requirements, all communication between the implemented tool and target application occurs over HTTP (protocol) using the HTTP requests as defined in the section 3. Python *requests*[3] library is used for managing these requests.

The Figure 4 describes the current class hierarchy of the implemented tool. An application entry point is within the *ScannerTool* class that has one to many relationship with module classes.

Each module class represents a module that is responsible for the part of the tool's functionality — as a concrete example can serve class *discover_xss* that belongs to the module that tries to discover reflected XSS on the target. A different example is *secret_finder* class that is responsible for identifying high entropy strings in the application source code.

Every module can be further decomposed into one to three submodules that serve for collecting information about the target (*Collector* class in the class diagram), processing collected information and artifacts (*Processor*) or presenting results (*Presenter*).

Modules can be dependent on another and can provide their results to other modules if requested. Collector submodules realize data acquisition done in the *OSINT Scan* phase. Processor submodules encapsulate functionality to process acquired data into *non-parsable* artifacts as well as to parse *parsable* artifacts received from other depending modules.

In the following paragraphs, some of the more challenging parts of the implementation are briefly described.

[3] See http://docs.python-requests.org

```
D:\Documents\Excel@FIT2019>python PoC.py https://target-application.local/

 Non-Destructive Penetration Testing Tool v0.1
 Author: Daniel Dusek (xdusek21@stud.fit.vutbr.cz)

 Non-destructive security scan initiated.
   Target: https://target-application.local/
   Using modules:
     |-> SiteMapper: Enabled
     |-> SourceCopier: Enabled
     |-> SecretUrlFinder: Enabled
     |-> ParamMiner: Enabled
     |-> PastebinTracker: Disabled
     |-> VulnLibraryFinder: Enabled
     |-> XSSDetector: Enabled


   Progress |                    | [53%]
   Current phase: OSINT SCAN
   Requests sent: 2575
   Elapsed time: 00:32:01
```

Figure 3. Example of the terminal output generated by the implemented tool.

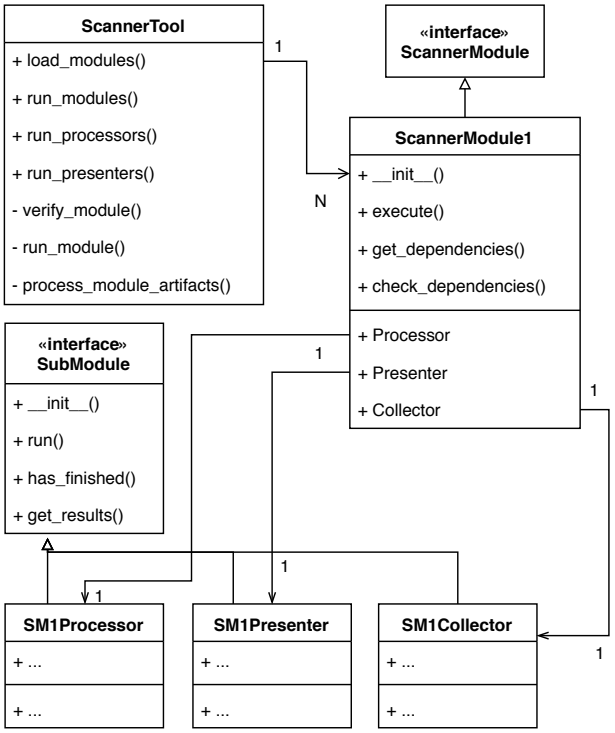


Figure 4. Class diagram illustrates relationship between classes of the implemented tool. Basic class *ScannerTool* has one to many relationship with *ScannerModule1* class which represents module written for the specific purpose and implements *ScannerModule* interface. Typically, every module has between one and three submodules for collecting information (*Collector*), processing it and creating artifacts (*Processor*) and presenting it (*Presenter*) — these are represented by accordingly named classes implementing the *SubModule* interface.

Monitoring Side Channels

When tested application belongs to a larger organization with many employees I believe it may be beneficial to assess what are the existing side channels used by the employees and try to monitor those for sensitive or valuable information. Example of such channels that I have identified can be public online forums and boards or when the company operates in the field of software development, then sites such as *Pastebin*[4] or *GitHub*[5].

Implemented tool enables a penetration tester to set up continuous monitoring of the mentioned *Pastebin* service for certain keywords that appear in the user submitted fragments of code. When such keyword is detected in the newly submitted fragment, a copy of the fragment is created and a tester is notified. Choosing the proper set of words to monitor for is a task that remains up to a tester during the OSINT Scan phase, **Define** step. Other tools that offer similar functionality exist and their variations have been developed in the past, an example of two such tools maybe be *pystemon*[6] and *pastemon*[7].

Direct and Indirect Requests

Sometimes information about target application can be acquired without directly interacting with it. There are services like *Web Archive*[8] that make a copy of the website at a certain point in time. *Web Archive* offers API for searching their copy of the site. Through this

[4] See https://pastebin.com
[5] See https://github.com
[6] See: https://github.com/cvandeplas/pystemon
[7] See https://github.com/xme/pastemon
[8] See https://web.archive.org/

API 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 the expected number of requests on the application is high and rate-limits apply.

Implemented tool takes advantage of this as well, for example when detecting existing URLs that are no longer accessible through navigating across the target application (potentially secret pages).

Tool Evaluation

The current implementation was tested and evaluated only on a subset of provided functionality. Results are presented below.

Use of vulnerable JavaScript library — For evaluation purposes, sample web application was prepared and used. Sample application included 10 currently known vulnerable versions of *jQuery*[9] library, 2 at the time of writing not vulnerable versions of the same library and 3 custom libraries without known vulnerability status. Additionally, real world web application[10] *davidriha.cz* was evaluated as well.

Table 1. Results of evaluation for detection of use of vulnerable JavaScript library. Column names *V, N, U* stands for *Vulnerable*, *Not Vulnerable* and *Unknown* detection status.

	Expected			Actual		
Application	V	N	U	V	N	U
Sample Application	10	2	3	10	2	3
davidriha.cz	1	0	1	1	0	1

Results testify that the application is capable of properly detecting the use of vulnerable *jQuery* library versions, as well as the use of not vulnerable libraries. It also suggests that there is a room for improvement in detecting vulnerability status of previously unknown libraries — e.g. at least through the means of static analysis to discover low-hanging-fruit vulnerabilities.

Listing Secrets and Access Tokens — Again, the sample application containing hidden secrets and tokens was prepared in order to evaluate the tool's ability to discover them. Table 2 displays length of hidden tokens an how well the implemented tool performed in discovering them.

Results recorded in Table 2 reflect that the tool performs great when discovering longer secrets and access tokens and underperforms when searching for the shorter ones. This is mainly because the searching algorithm is based on looking for the high-entropy strings — typical access token or secret is generated with specific requirements on its entropy level. Shorter strings, such as *type=checkbox* (common string appearing in HTML source) has high entropy while not being exactly considered a secret. This could be improved by implementing additional heuristics for the context-based search for the shorter tokens while keeping the high-entropy string search approach for longer strings.

Table 2. Table presents how well the implemented tool performed in discovering hidden tokens in sample application. Token length is measured in characters.

Token Length	Hidden	Discovered	False Positives
10	10	10	216
15	10	10	113
20	10	10	3
30	10	10	1
50	10	10	0

Real-time monitoring of side channels — To evaluate the tool's ability to monitor side channels, an experiment using the *Pastebin* service was conducted.

First, a utility capable of posting a new code snippet to the mentioned service was created. This tool was then set up to post 200 public snippets over the course of 24 hours. These snippets contained random lines of code and always included keywords that the implemented tool was monitoring. Some of the created posts were pseudo-randomly set to be deleted in the pseudo-randomly selected time period after their creation.

Then the implemented tool to monitor side channels was started for the same period of time and set up to store snippets containing keywords of interest.

Finally, after 24 hours, the number of stored snippets was compared to the number of snippets generated by the evaluation tool. Implemented tool managed to store 188 snippets generated by the evaluation tool. The 12 snippets that escaped the monitoring were deleted in the 120 seconds window after their creation. This behavior is caused by the 2 minutes delay between new posts being available through the *Pastebin* scraping API — if the post does not exist 2 minutes after being created, it will never be published through the API.

That leads me to the conclusion that this tool is currently capable of near-real-time *Pastebin* side-channel monitoring. While it would be possible to increase the tool's effectiveness by scraping directly from the *Pastebin*'s front site, it would mean knowingly breaching the *Pastebin*'s terms of service and therefore this

[9] See https://jquery.com/

[10] Consent was acquired from the application maintainer prior to evaluation, available online http://davidriha.cz

extension will not be realized.

5. Conclusions

In this paper, I proposed a new approach to web application penetration testing that focuses on *non-destructive* and *non-permanent* interactions with the target application. This approach supports automation and when followed, allows for the creation of automated utilities that can be run against the production environment without the fear of breaking it. Later on in the paper, I present the proof of concept tool that I have implemented to demonstrate the possibility of applying proposed principles in practice and to support the approach with an evidence of it.

The implemented tool is capable of penetration testing real production systems without the risk of causing downtime. Furthermore, it automates a lot of repetitive manual tasks that a penetration tester would have to execute and therefore saves time and frees their hands to focus on other, possibly more complex and hard to automate testing tasks.

Evaluation of the tool that has been conducted so far confirms that the tool is well-capable of discovering vulnerable versions of JavaScript libraries being used by target application both in the simulated and real-world scenarios. It is also quite capable of detecting high-entropy-based secrets and access tokens that are longer than 20 characters. Lower length secrets and access tokens cannot be effectively detected by the current algorithm. The evaluation of the real-time side channel monitoring shown that the tool is capable of detecting 188 of 200 messages of interest. The twelve messages that were lost were lost due to the delayed response from the monitored service API — this cannot be bypassed without breaching the service's terms of use, but it is technically possible.

The tool presented in the latter part of this paper is still being actively worked on. In the future, additional features such as the possibility of using a proxy (including *socks5 proxy*) for outgoing requests or monitoring domain registration feed for typosquatting domain registration will be added. One additional notable extension would be the possibility to offload certain tasks that offer space for parallelism to other programs not implemented in the python, as the python is notoriously known for the parallelism inefficiency.

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Jan Pluskal for his help and guidance through the whole process of working on my master thesis which this article is based on.

References

[1] Graham Cormode and Balachander Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), 2008.

[2] Trends in consumer mobility report. Technical Report 1.415.913.4416, Bank of America, August 2017.

[3] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[4] Richard Barber. Hackers profiled—who are they and what are their motivations? *Computer Fraud & Security*, 2001(2):14–17, 2001.

[5] Andrew Simmonds, Peter Sandilands, and Louis Van Ekert. An ontology for network security attacks. In *Asian Applied Computing Conference*, pages 317–323. Springer, 2004.

[6] Veronica Valeros. Make it count: an analysis of a brute-forcing botnet. *The Journal on Cybercrime & Digital Investigations*, 1(1), 2016.

[7] Catalin Cimpanu. Gamarue botnet uses hijacked wordpress sites to send spam with js payloads, April 2016, (accessed November 21, 2018).

[8] Patrick Engebretson. *The Basics of Hacking and Penetration Testing: Ethical Hacking and Penetration Testing Made Easy*. Syngress Publishing, 2nd edition, 2013.

[9] OWASP. About the open web application security project, September 2018, (accessed November 25, 2018).

[10] A Muller, M Meucci, E Keary, and D Cuthbert. Owasp testing guide 4.0, 2014 (accessed November 21, 2018).

[11] R. Fielding and J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014.

[12] Netsparker. Netsparker web application security scanner benefits overview — netsparker, 2019 (accessed April 5, 2019).

[13] Portswigger. Auditing, 2019 (accessed April 5, 2019).