Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners

Andrea Valenza
University of Genova
andrea.valenza@dibris.unige.it

Gabriele Costa
IMT School for Advanced Studies Lucca
gabriele.costa@imtlucca.it

Alessandro Armando
University of Genova
alessandro.armando@unige.it

Abstract

The first step of every attack is reconnaissance, i.e., to acquire information about the target. A common belief is that there is almost no risk in scanning a target from a remote location. In this paper we falsify this belief by showing that scanners are exposed to the same risks as their targets. Our methodology is based on a novel attacker model where the scan author becomes the victim of a counter-strike. We developed a working prototype, called RevOK, and we applied it to 78 scanning systems. Out of them, 36 were found vulnerable to XSS. Remarkably, RevOK also found a severe vulnerability in Metasploit Pro, a mainstream penetration testing tool.

1 Introduction

Performing a network scan of a target system is a surprisingly frequent operation. There can be several agents behind a scan, e.g., attackers that gather technical information, penetration testers searching for vulnerabilities, Internet users checking a suspicious address. Often, when the motivations of the scan author are unknown, it is perceived by the target as a hostile operation. However, scanning is so frequent that it is largely tolerated by the target. Even from the perspective of the scanning agent, starting a scan seems not risky. Although not completely stealthy, an attacker can be reasonably sure to remain anonymous by adopting basic precautions, such as proxies, virtual private networks and onion routing.

Yet, expecting an acquiescent scan target is a mere assumption. The scanning system may receive poisoned responses aiming to trigger vulnerabilities in the scanning host. Since most scanning systems generate an HTML report, scan authors can be exposed to attacks via their browser. This occurs when the scanning system permits an unsanitized flow of information from the response to the user browser. To illustrate, consider the following, minimal HTTP response.

```
HTTP/1.1 200 OK
Server: nginx/1.17.0
...
```

A naive scanning system might extract the value of the `Server` field (namely, the string `nginx/1.17.0` in the above example) and include it in the HTML report. This implicitly allows the scan target to access the scan author's browser and inject malicious payloads.

In this paper we investigate this attack scenario. We start by defining an attacker model that precisely characterizes the threats informally introduced above. To the best of our knowledge, this is the first time that such an attacker model is defined in literature. Inspired by the attacker model, we define an effective methodology to discover cross-site scripting (XSS) vulnerabilities in the scanning systems and we implement a working prototype. We applied our prototype to 78 real-world scanning systems. The results confirm our expectation: several (36) scanning systems convey attacks. All of these vulnerabilities have been notified through a responsible disclosure process.

The most remarkable outcome of our activity is possibly an XSS vulnerability enabling remote code execution (RCE) in Rapid7 Metasploit Pro. We show that the attack leads to the complete takeover of the scanning host. Our notification prompted Rapid7 to undertake a wider assessment of their products based on our attacker model.

The main contributions of this paper are:

1. a novel attacker model affecting scanning systems;
2. a testing methodology for finding vulnerabilities in scanning systems;
3. RevOK, a prototype implementation of our testing methodology;
4. an analysis of the experimental results on 78 real-world scanning systems, and;
5. three application scenarios highlighting the impact of our attacker model.

This paper is structured as follows. Section 2 recalls some preliminary notions. Section 3 presents our attacker model.

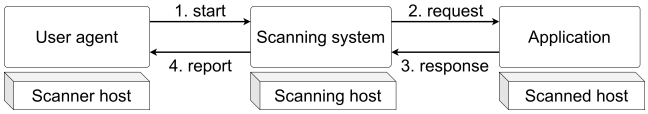

Figure 1: Abstract architecture of a scanning system.

We introduce our methodology in Section 4. Our prototype and experimental results are given in Section 5. Then, we present the three use cases in Section 6, while we survey on the related literature in Section 7. Finally, Section 8 concludes the paper.

2 Background

In this section we recall some preliminary notions necessary to correctly understand our methodology.

2.1 Scanning systems

A scanning system is a piece of software that (i) stimulates a target through network requests, (ii) collects the responses, and (iii) compiles a report. Security analysts often use scanning systems for technical information gathering [8]. Scanning systems used for this purpose are called *security scanners*. Our definition encompasses a wide range of systems, from complex vulnerability scanners to simple ping utilities.

Figure 1 shows the key actors involved in a scan process. Human analysts use a user agent, e.g., a web browser, to select a target, possibly setting some parameters, and start the scan (1. start). Then, the scanning system crafts and sends request messages to the target (2. request). The scanning system parses the received response messages (3. response), extracts the relevant information and provides the analyst with the scan result (4. report). Finally, the analysts inspect the report via their user agent.

Whenever a scanning system runs on a separate, remote scanning host, we say that it is provided *as-a-service*. Instead, when the scanner and scanning hosts coincide, we say that the scanning system is *on-premise*.

A popular, command line scanning system is Nmap [25]. To start a scan, the analyst runs a command from the command line, such as

```
nmap -sV 172.16.1.26 -oX report.xml
```

Then, Nmap scans the target (`172.16.1.26`) with requests aimed at identifying its active services (`-sV`). By default, Nmap sends requests to 1,000 frequently used TCP ports and collects responses from the services running on the target. The result of the scan is then saved (`-oX`) on `report.xml`. Interestingly, some web applications, e.g., Nmap Online [15], provide the functionalities of Nmap as-a-service.

Scanning systems are often components of larger, more complex systems, sometimes providing a browser-based GUI. For instance, Rapid7 Metasploit Pro is a full-fledged penetration testing software. Among its many functionalities, Metasploit Pro also performs automated information gathering, even including vulnerability scanning. The reporting system of Metasploit Pro is based on an interactive Web UI used to browse the report.

2.2 Taint analysis

Taint analysis [26] refers to the techniques used to detect how the information flows within a program. Programs read inputs from some sources, e.g., files, and write outputs to some destinations, e.g., network connections. For instance, taint analysis is used to understand whether an attacker can force a program to generate undesired/illegal outputs by manipulating some of its inputs. A *tainted flow* occurs when (part of) the input provided by the attacker is included in the (tainted) output of the program. In this way, the attacker controls the tainted output which can be used to inject malicious payloads to the output recipient.

2.3 Cross-site scripting

Cross-site scripting (XSS) is a major attack vector for the web, stably in the OWASP Top 10 vulnerabilities [12] since its initial release in 2003. Briefly, an XSS attack occurs when the attacker injects a third-party web page with an executable script, e.g., a JavaScript fragment. The script is then executed by the victim's browser. The simplest payload for showing that a web application suffers from an XSS vulnerability is

```
<script>alert(1)</script>
```

that causes the browser to display an alert window. This payload is often used as a proof-of-concept (PoC) to safely prove the existence of an XSS vulnerability.

There are several variants to XSS. Among them, *stored* XSS has highly disruptive potential. An attacker can exploit a stored XSS on a vulnerable web application to permanently save the malicious payload on the server. In this way, the attack is directly conveyed by the server that delivers the injected web page to all of its clients. Another variant is *blind* XSS, in which the attacker cannot observe the injected page. For this reason, blind XSS relies on a few payloads, each adapting to multiple HTML contexts. These payloads are called *polyglots*. A remarkable example is the polyglot presented in [11] which adapts to at least 26 different contexts.

3 Attacker model

The idea behind our attacker model is sketched in Figure 2 (bottom), where we compare it with a traditional web security attacker model (top). Typically, attackers use a security scanner to gather technical information about a target application. If the application suffers from some vulnerabilities,

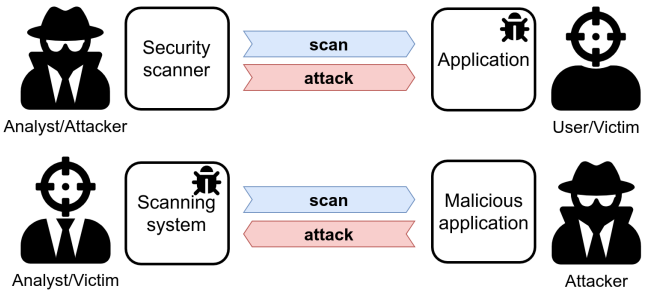

Figure 2: Comparison between attacker models.

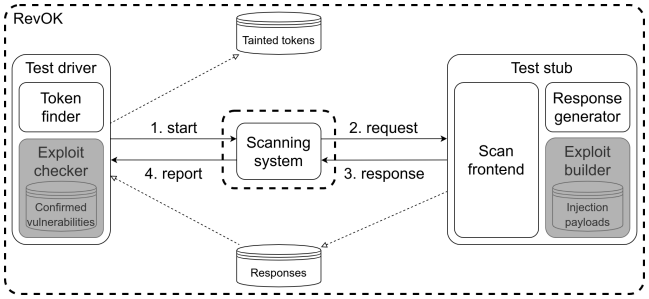

Figure 3: Phase 1 – find tainted flows.

attackers can exploit them to deliver an attack towards their victims, e.g., the application users. On the contrary, in our attacker model attackers use malicious applications to attack the author of a scan, e.g., a security analyst.

Here are the two novelties of our attacker model.

1. Attacks are delivered through HTTP *responses* instead of *requests*.

2. Attackers exploit the vulnerabilities of scanning systems to strike their victims, i.e., the scan initiator.

Below, we detail the attacker's goal and capabilities.

Attacker goal. The objective of the attacker is to directly strike the analyst. To do so, the attacker exploits the vulnerabilities of the target scanning system and its reporting system to hit the analyst user agent. In this work, we assume that the user agent is a web browser. This assumption covers every as-a-service scanning system, as well as many on-premise ones, which generate HTML reports. As a consequence, here we focus on XSS which is a major attack vector for web browsers. As usual in XSS, the attacker succeeds when the victim's browser executes a piece of attacker-provided code, e.g., JavaScript.

Attacker capabilities. First, we state that the attacker has adequate resources to detect vulnerabilities in scanning systems before deploying the malicious application. However, the attacker capabilities do not include the possibility of observing the internal logic of the scanning system. That is, our attacker operates in black-box mode.

Secondly, our attacker has complete control over the malicious application, e.g., the attacker owns the scanned host. However, we do not assume that the attacker can force the victim to initiate the scanning process.

4 Testing methodology

In this section, we define a vulnerability detection methodology based on our attacker model.

4.1 Test execution environment

Our methodology relies on a *test execution environment* (TEE) to automatically detect vulnerabilities in scanning systems. In particular, a *test driver* simulates the user agent of the security analyst, while a *test stub* simulates the scanned application. Our TEE can (i) start a new scan, (ii) receives the requests of the scanning system, (iii) craft the responses of the target application, and (iv) access the report of the scanning system. Intuitively, the TEE replicates the configuration of Figure 1. In this configuration, the test driver is executed by the scanner host, and the test stub runs on the scanned host. In general, the test driver is customized for each scanning system under testing. For instance, it may consist of a Selenium-enabled [14] browser stimulating the web UI of the scanning system.

Both the test driver and the test stub consist of some submodules. These submodules are responsible for implementing the two phases described below.

4.2 Phase 1: tainted flows enumeration

The first phase aims at detecting the existing tainted destinations in the report generated by the scanning system. Having a characterization of the tainted flows is crucial to deal with the input transformation logic of the target scanning system. In general, since payloads may be arbitrarily modified before being displayed in the report, detecting actual injections is non-trivial. Instead, through this phase, injections can be detected just by monitoring tainted destinations. The process is depicted in Figure 3. Initially, the test driver asks the scanning system to perform a scan of the test stub. The scan logic is not exposed by the scanning system and, thus, it is opaque from our perspective. Nevertheless, it generates some requests toward the test stub. Each request is received by the *scan frontend* and dispatched to the *response generator*, which crafts the response.

The response generation process requires special attention. One might think that a single, general-purpose response is sufficient. However, some scanning systems process the responses in non-trivial ways. For instance, they may abort the scan if a malformed or suspicious response is received. For this reason, we proceed as follows. First, we generate a

response template, i.e., an HTTP response containing variables, denoted by t. Response templates are generated from a fuzzer through a *probabilistic context-free grammar* (PCFG). A PCFG is a tuple (N, Σ, R, S, P), where $G = (N, \Sigma, R, S)$ is a context-free grammar such that N is the set of non-terminal symbols, Σ is the set of terminal symbols, R are the production rules and S is the starting symbol. The additional component of the PCFG, namely $P : R \to [0, 1]$, associates each rule in R with a probability, i.e., the probability to be selected by the fuzzer generating a string of G. Additionally, we require that P is a probability distribution over each non-terminal α, in symbols

$$\forall \alpha \in N. \sum_{(\alpha \mapsto \beta) \in R} P(\alpha \mapsto \beta) = 1$$

In the following, we write $\alpha \mapsto_p \beta$ for $P(\alpha \mapsto \beta) = p$ and $\alpha \mapsto_{p_1} \beta_1 |_{p_2} \ldots |_{p_n} \beta_n$ for $\alpha \mapsto_{p_1} \beta_1, \ldots, \alpha \mapsto_{p_n} \beta_n, \alpha \mapsto_{p_e}$ `""` (where `""` is the empty string).

The probability values appearing in our PCFG are assigned according to the results presented in [20, 21]. There, the authors provide a statistical analysis of the frequency of real response headers as well as a list of information-revealing ones. Such headers are thus likely to be reported by a scanning system. Finally, when the frequency of a field is not given (e.g., for variables), we apply the uniform distribution.

An excerpt of our PCFG is given in Figure 4. For the sake of presentation, here we omit some of the rules and we refer the interested reader to the project web site[1]. The grammar defines the structure of a generic HTTP response (`Resp`) made of a version (`Vers`), a status (`Stat`), a list of headers (`Head`), and a body (`Body`). Variables t are all fresh and they can appear in several parts of the generated response template. In particular, variables can be located in status messages (i.e., `Succ`, `Redr`, `ClEr` and `SvEr`), header fields (i.e., `Serv`, `PwBy`, `Locn`, `SetC`, `CntT`, `AspV`, `MvcV`, `Varn`, `StTS`, `CnSP`, `XSSP` and `FrOp`) and body. For instance, a field can be `Server: nginx/`t, where `nginx/` is a server type (`SrvT`, omitted for brevity).

The response template is then populated by replacing each variable with a *token*. A token is a unique sequence of characters that is both *recognizable*, i.e., it has a negligible probability of appearing by chance, and *uninterpreted*, i.e., the browser treats it as plain text, when appearing in an HTML document. All tokens are mapped to the responses containing them. Responses are stored in a database. Finally, the test driver matches the tokens appearing in the responses database with those occurring in the scan report. Such tokens are evidence that there are tainted flows in the internal logic of the scanning system. Tokens mark the source and the sink of a flow in the response and report, respectively. All these tokens are stored in the tainted tokens database.

```
Resp ↦1 Vers Stat Head Body
Vers ↦0.5 "HTTP/1.0" |0.5 "HTTP/1.1"
Stat ↦0.554 Succ |0.427 Redr |0.013 ClEr |0.006 SvEr
Succ ↦0.5 "200 OK" |0.5 "200" t
Redr ↦0.386 "301 Moved Permanently" |0.386 "301" t
      |0.114 "302 Found" |0.114 "302" t
ClEr ↦0.26 "403 Forbidden" |0.26 "403" t
      |0.24 "404 Not Found" |0.24 "404" t
SvEr ↦0.5 "500 Internal Server Error" |0.5 "500" t
Head ↦1 Serv PwBy Locn SetC CntT AspV MvcV Varn
        ↪ StTS CnSP XSSP FrOp
Serv ↦0.475 "Server:" t |0.475 "Server:" SrvT t
PwBy ↦0.24 "X-Powered-By: php"
      |0.24 "X-Powered-By:" t
Locn ↦0.315 "Location:" Link |0.315 "Location:" t
Link ↦0.516 "https://" t
      |0.167 "http://" t ":8899"
      |0.135 "http://" t ":8090"
      |0.065 "http://" t "/login.lp"
      |0.059 "/nocookies.html"
      |0.058 "cookiechecker?uri=/"
SetC ↦0.175 "Set-Cookie:" Ckie
Ckie ↦0.471 "__cfduid=" t |0.394 "PHPSESSID=" t
      |0.087 "ASP.NET_Session=" t
      |0.048 "JSESSIONID=" t
CntT ↦0.07 "X-Content-Type-Options: nosniff"
      |0.07 "X-Content-Type-Options:" t
AspV ↦0.5 "X-AspNet-Version:" t
MvcV ↦0.5 "X-AspNetMvc-Version:" t
Varn ↦0.5 "X-Varnish:" t
StTS ↦0.5 "Strict-Transport-Security:" STSA
STSA ↦0.111 "max-age=" ℕ+
      |0.111 "max-age=" t
      |0.111 "max-age=" ℕ+ "; preload"
      |0.111 "max-age=" t "; preload"
```

Figure 4: Response template grammar (excerpt).

4.3 Phase 2: vulnerable flows identification

The second phase aims to confirm which tainted flows are actually vulnerable. We use PoC exploits to confirm the vulnerability. The workflow is depicted in Figure 5. As for the first phase, the test driver launches a scan of the test stub. When the test stub receives the requests, the exploit builder extracts a response from the responses database. Then, the response is injected with a PoC exploit. More precisely, a tainted token is selected among those generated during Phase 1. The tainted token in the response is replaced with a payload taken from a predefined injection payload database. In general, a vulnerability is confirmed by the test driver according to predefined, exploit-dependent heuristics. Although tainted flows can be subject to different types of vulnerabilities, as discussed in Section 3, we focus on XSS. Thus, the heuristics implemented by the exploit checker consists of recognizing a vulnerable flow when an alert window is spawned by the corresponding, tainted flow. Finally, the exploit checker stores the vulnerable flows in the confirmed vulnerabilities database.

The definition of injection payload is non-trivial. Since our

[1] https://github.com/AvalZ/RevOK

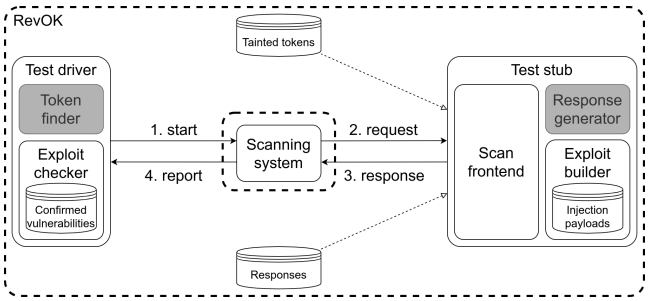

Figure 5: Phase 2 – find vulnerable flows.

TEE applies to both on-premise and as-a-service scanning systems, some issues must be considered. The first issue is testing performances. As a matter of fact, scanning systems can take a considerable amount of time to perform a single scan. Moreover, as-a-service scanning systems should not be flooded by requests to avoid degradation of the quality of service. For these reasons, we aim to limit the number of payloads to check.

As discussed in Section 2.3, polyglots allow us to test multiple contexts with a single payload. In this way, we increase the success probability of each payload and, thus, we reduce the overall number of tests.

In principle, we might resort to the polyglot of [11], which escapes 26 contexts. However, its length (144 characters) is not adequate since many scanning systems shorten long strings when compiling their reports, so preventing the exploit from taking place. To avoid this issue, we opted for polyglots such as `"'/><img src='x' onerror='alert(1)'/>`. This is rendered by the browser when appearing inside both an HTML tag and an HTML attribute. The reason is that the initial `"` and `'` allow the payload to escape from quoted attributes.

Furthermore, delivering the JavaScript payload in `onerror` has two advantages. First, it circumvents basic input filtering methods, e.g., blacklisting of the `script` string. Secondly, our payload applies to both static and dynamic reports. More precisely, a static report consists of HTML pages that are created by the scanning system and subsequently loaded by the analyst's browser. Instead, a dynamic report is loaded by the browser and updated by the scanning system during the scan process. The HTML5 standard specification [16, § 8.4.3] clearly states that browsers can skip the execution of dynamically loaded scripts. For this reason, our payload binds the script execution to an *error* event that we trigger using a broken image link (i.e., `src='x'`). A concrete example of this scenario is discussed in Section 6.2.

5 Implementation and results

In this section, we present our prototype RevOK. We used it to carry out an experimental assessment that we discuss in Section 5.3.

5.1 RevOK

Our prototype consists of two modules: the test driver and the test stub. We detail them below.

Test driver A dedicated test driver is used for each scanning system. The test driver (i) triggers a scan against the test stub, (ii) saves the report in HTML format and (iii) processes the report to detect tainted and vulnerable flows (in Phase 1 and 2, respectively). While (iii) is the same for all the scanning systems, (i) and (ii) may vary.

In general, the implementation of (i) and (ii) belongs to two categories depending on whether the scanning system has a programmable interface or only a GUI. When a programmable interface is available, we implement a Python 3 client application. For instance, we use the native *os* Python module to launch Nmap so that its report is saved in a specific location (as describe in Section 2). Similarly, we use the *requests* Python library[2] to invoke the REST APIs provided by a scanning system and save the returned HTML report. Instead, when the scanning system only supports GUI-based interactions, we resort to GUI automation. In particular, we use the Selenium Python library[3] for browser-based GUIs and *PyAutoGUI*[4] for desktop GUIs. In the case of GUI automation, the test driver repeats a sequence of operations recorded during a manual test.

Finally, for the report processing step (iii) we distinguish between two operations. The tainted flow detection trivially searches the report for the injected tokens provided by the response generator (see below). Instead, vulnerable flows are confirmed by checking the presence of alert windows through the Selenium function `switch_to_alert()`.

Test stub For the response generator, we implemented the PCFG grammar fuzzer detailed in Section 4.2 in Python. Tokens are represented by randomly-generated Universally Unique Identifiers [22] (UUID). A UUID consists of 32 hexadecimal characters organized in 5 groups that are separated by the - symbol. An example UUID is `018d54ae-b0d3-4e89-aa32-6f5106e00683`. As required in Section 4.2, UUIDs are both recognizable (as collisions are extremely unlikely to happen) and uninterpreted (as they contain no HTML special characters).

On the other hand, starting from a response, the exploit builder replaces a given UUID with an injection payload. Payloads are taken for a predefined list of selected polyglots, as discussed in Section 4.3.

[2] https://requests.readthedocs.io
[3] https://selenium-python.readthedocs.io/
[4] https://pyautogui.readthedocs.io

5.2 Selection criteria

We applied our prototype implementation to 78 scanning systems. The full list of scanning systems, together with our experimental results (see Section 5.3), is given in Table 1. There, we use 🌍 and ⬇ to distinguish between as-a-service and on-premise scanning systems, respectively.

For our experiments, we searched for scanning systems included in several categories. In particular, we considered security scanners, server fingerprinting tools, search engine optimization (SEO) tools, redirect checkers, and more. From these, we removed scanning systems belonging to the following categories.

- Abandonware, i.e., on-premise scanning systems that were not maintained in the last 5 years.
- Paywalled, i.e., scanning systems that are not free and have no trial version.
- Scheduled, i.e., as-a-service scanning systems that only perform periodic scans, not controlled by the analyst.

5.3 Results

We applied RevOK to the scanning systems of Table 1. For each scanning system, we used RevOK to execute 10 scan rounds (see Section 4) and we listed all the detected tainted and vulnerable flows. As a result, we discovered that 67 scanning systems have tainted flows and, among them, 36 are vulnerable to XSS.

In Table 1, for each scanning system we report the number of tainted and vulnerable flows (**T** and **V**, respectively) detected by RevOK. After running RevOK, we also conducted a manual vulnerability assessment of each scanning system. The assessment consisted of a review of each tainted flow, followed by a manual payload generation (see below).

Under column **M**, ✓ indicates that an XSS vulnerability was found by a human analyst starting from the outcome of RevOK. It is worth noticing that only in one case, i.e., DupliChecker, RevOK resulted in a false negative w.r.t. the manual analysis. By investigating the causes, we discovered that DupliChecker performs URL encoding on the tainted locations. This encoding, among other operations, replaces white spaces with `%20`, thus invalidating our payloads. To effectively bypass URL encoding, we replaced white spaces (U+0020) with non-breaking spaces (U+00A0) that are not modified. Thus, we defined a new polyglot payload that uses non-breaking spaces and we added it to the injection list included in RevOK. Using this new payload, RevOK could also detect the vulnerability in DupliChecker.

At the time of writing, all the vulnerabilities detected by RevOK have been reported to the tool vendors and are undergoing a responsible disclosure process (see Appendix A).

In Figure 6 we show the frequency of the tainted and vulnerable flows over the 14 fields considered by RevOK. Location

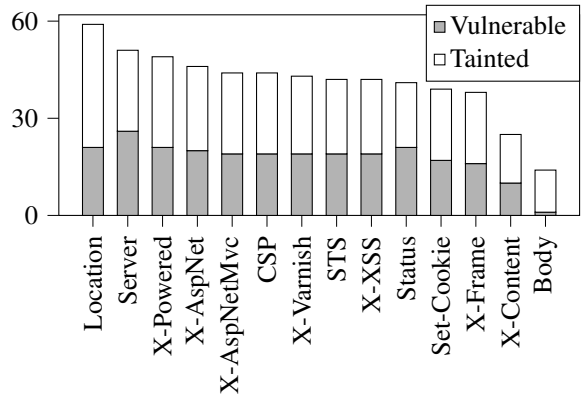

Figure 6: Frequency of tainted and vulnerable flows.

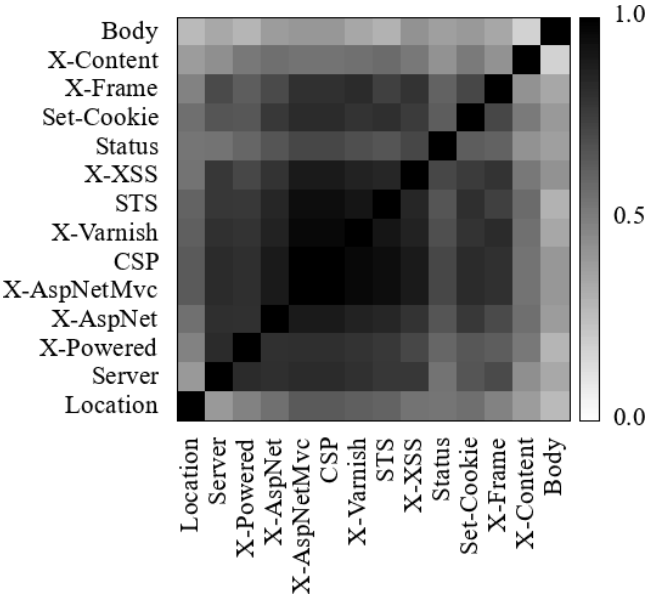

Figure 7: Correlation between tainted fields.

has 59 tainted flows, the highest number, and 21 vulnerable flows. Server only has 51 tainted flows, but it has 26 vulnerable flows, the highest number. On the other hand, Body has only 14 tainted flows and only 1 vulnerable flow. This highlights that most scanning systems sanitize the Body field in their reports. The reason is that HTTP responses most likely contain HTML code in their Body. Thus, sanitization is mandatory to preserve the report layout. Also, the Body field is often omitted by the considered scanning systems.

In Figure 7 and Figure 8 we show the correlation matrices for tainted and vulnerable fields, respectively. From these matrices we observe a few, relevant facts. We briefly discuss them below.

The first observation is that the Body field is almost unrelated to the other fields, both in terms of tainted and vulnerable flows. This is somehow expected since the Body field is often neglected as discussed above.

Table 1: Experimental results for the considered scanning systems. († requested to stay anonymous.)
T is the number of tainted flows, V is the number of vulnerable flows, M indicates if a human analyst found a vulnerability.

Name	T	V	M	Name	T	V	M	Name	T	V	M
AddMe	11	11	✓	InternetOfficer	2	1	✓	Security Headers	13	-	
AdResults	14	-		[*Anonymous*]†	11	1	✓	SEO Review Tools	-	-	
Arachni	14	-		iplocation.net	-	-		SeoBook	12	11	✓
AUKSEO	-	-		IPv6 Scanner	-	-		SERP-Eye	-	-	
BeautifyTools	13	-		itEXPERsT	-	-		Server Headers	13	12	✓
BrowserSPY	9	-		IVRE	2	-		Site 24x7	13	13	✓
CheckHost	1	-		JoydeepDeb	13	13	✓	SQLMap Scanner	1	1	✓
CheckMyHeaders.com	1	-		JSON Formatter	13	13	✓	SSL Certificate Tools	12	-	
CheckSERP	11	-		LucasZ ZeleznY	2	1	✓	StepForth	12	11	✓
CheckShortURL	1	1	✓	Metasploit Pro	11	3	✓	StraightNorth	-	-	
Cloxy Tools	11	-		Monitor Backlinks	12	-		SubnetOnline	14	13	✓
CookieLaw	1	-		Nessus	11	-		Sucuri Site Check	3	-	
CookieMetrix	2	1	✓	Nikto Online	2	2	✓	SureOak	9	8	✓
DNS Checker	1	1	✓	Nmap	14	-		TheSEOTools	1	1	✓
DNSTools	-	-		Nmap Online	12	1	✓	Tutorialspots	13	13	✓
Dupli Checker	1	-	✓	Online SEO Tools	12	12	✓	Url X-Ray	1	-	
evilacid.com	12	12	✓	OpenVAS	3	-		Urlcheckr	10	-	
expandUrl	1	-		OWASP ZAP	4	-		Urlex	-	-	
FreeDirectoryWebsites	13	13	✓	Pentest-Tools	2	1	✓	w-e-b.site	13	13	✓
GDPR Cookie Scan	-	-		Port Checker	10	-		W3dt.Net	12	11	✓
GeekFlare	12	-		Redirect Check	11	10	✓	Web Port Scanner	-	-	
Hacker Target	13	-		Redirect Detective	2	-	✓	Web Sniffer	14	-	
HTTP Tools	12	12	✓	ReqBin	13	-		WebConfs	13	12	✓
httpstatus.io	14	-		Resplace	12	-		WebMap	14	1	✓
InsightVM	3	-		RexSwain.com	13	1	✓	What Is My IP	12	-	
InternetMarketingNinjas	1	-		Search Engine Reports	1	1	✓	WMap	12	10	✓

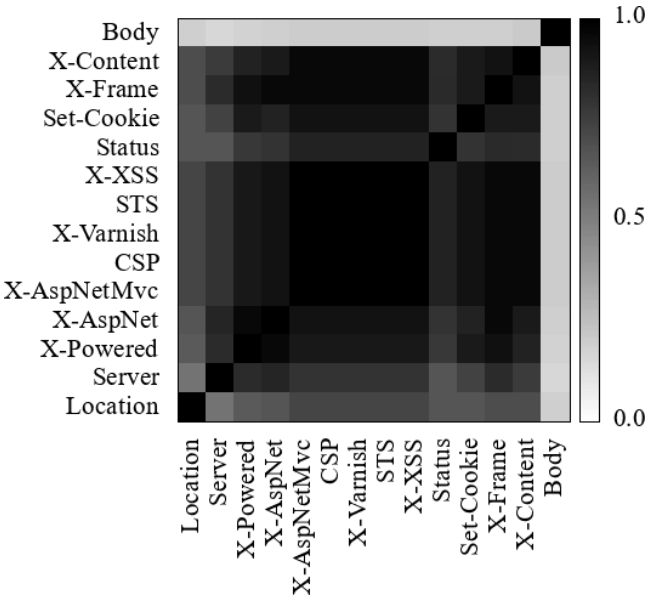

Figure 8: Correlation between vulnerable fields.

Also the Location field is weakly correlated with the other fields. This is due to the behavior of redirect checkers. As a matter of fact, this category of scanning systems focus on Location, and, in most cases, ignore the other fields. An in-depth evaluation of the behavior of redirect checkers is given in the application scenario of Section 6.3.

An argument similar to the previous one for Location also applies to Status Message. The Status Message is typically used by scanning systems that carry out availability checks, e.g., to verify that a web site is up and running.

Finally, for what concerns all the other fields, we observe an extremely strong correlation. The confirms the proposition of [20] about the security relevance of the headers that we are considering. Indeed, most of the scanning systems included in our experiments report them all. This also highlights that the exposure of the scanning systems is not field-dependent, e.g., when a scanning system is vulnerable via one these fields, most likely it is also vulnerable via the others.

6 Application Scenarios

In this section, we present three application scenarios for our methodology. For each scenario, we highlight the sub-class of vulnerable scanning systems, the vulnerability and

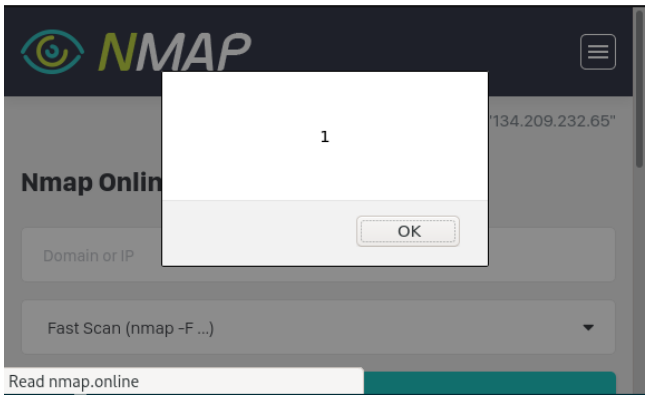

Figure 9: XSS PoC on Nmap Online.

its impact if an attacker were to use it in the wild. For each subclass of scanning systems, we chose a representative that we present as a concrete case study: Nmap Online for as-a-service scanning systems, Metasploit Pro for on-premise ones, and CheckShortURL [5] for redirect checkers.

6.1 Scan Attribution

Attack attribution is a hot topic since it is often difficult or even impossible to achieve. The main reasons are the structure of the network and some state-of-the-art technologies that enable clients anonymity. For instance, analysts can use proxies, virtual private networks, and onion routing to hide the actual source of the requests from the recipient. However, an injected browser may be forced to send identifying data directly inside the HTTP requests, so making network-level anonymization techniques ineffective. In this section, we show how to attribute scans using our attacker model through an application scenario based on Nmap Online [15].

Nmap Online vulnerability Nmap Online is a web application providing some of the functionalities of Nmap. Users can scan a target with Nmap without having to install it on their machine. Furthermore, since requests originate from the Nmap Online server, users can stay anonymous w.r.t. the scan target. When users start a scan, they select the target IP and the scan type. The Nmap Online website scans its target and displays the retrieved information to the user, e.g., server type and version.

Nmap Online reports suffered from an XSS vulnerability.[5] Figure 9 shows an injected report. The injection occurs on the Server response header. In this case, the Server field was set to `<script>alert(1)</script>`.

Browser hooking Since there is no guarantee that more than one scan will occur, we recur to browser hooking, which can be obtained with a single XSS payload. A hooked browser becomes the client in a command and control (C2) infrastructure, thus actively querying the C2 server for instructions. This allows the attacker to submit arbitrary commands afterward even when no other scans occur.

An effective way to achieve browser hooking is through BeEF [2]. In particular, the BeEF C2 client is injected via the script *hook.js*. For instance, we can deploy hook.js by setting the Server header to `<script src='http://[C2]/hook.js'></script>` where `[C2]` is the IP address of the C2 server.

Fingerprinting The BeEF framework includes modules[6] for fingerprinting the victim host. For instance, the *browser* module allows us to get the browser name, version, visited domains, and even starting a video streaming from the webcam. Similarly, the *host* module allows us to retrieve data such as physical location and operating system details. Some of these operations, e.g., browser fingerprinting, require no victim interaction. Instead, others need the victim to take some actions, e.g., explicitly grant permission to use the webcam. To overcome these hurdles, attackers usually employ auxiliary techniques, e.g., credential theft, implemented by some BeEF modules, e.g., *social engineering*. Finally, the overall fingerprinting process can be automated through the BeEF *autorun rule engine* [1].

6.2 Scanning host takeover

On-premise scanning systems, which run on the analyst's host, may have privileged, unrestricted access to the underlying platform. In some cases, on-premise systems are provided with a user interface that includes both the reporting system and a control panel. When such a user interface is browser-based, a malicious scan target can inject commands in the reporting system and perform lateral movements by triggering the scanning system controls.

The attack strategy abstractly described above must be implemented through concrete steps that are specific to the scanning system. In this section, we show an implementation of this attack strategy for the popular scanning system Metasploit Pro. In particular, we show how to perform lateral movements leading to a complete scanning host takeover through remote code execution (RCE). Finally, we carry out an impact evaluation.

CVE-2020-7354 and CVE-2020-7355 Metasploit Pro is a full-fledged penetration testing framework. It has a browser-based UI that integrates both a scan reporting system and many controls for running the most common tasks, including host scanning. Each command is executed by the Metasploit back-end, which is stimulated through a REST API.

[5] The vulnerability was fixed on March 24, 2020.

[6] https://github.com/beefproject/beef/wiki/BeEF-modules

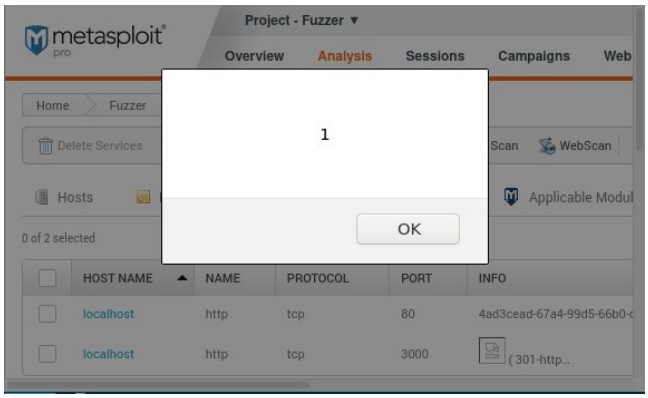

Figure 10: Stored XSS PoC on Metasploit Pro.

The vulnerability we found affects versions 4.17.0 and below. It was remediated on May 14, 2020, with patch 4.17.1.[7] A malicious scan target can inject a Stored XSS payload in the UI. Multiple pages are vulnerable, e.g., `/hosts/:id` and `/workspaces/:id/services`.

Metasploit Pro fetches the Server header and displays it inside the *INFO* column with no sanitization. Figure 10 shows the effect of setting the Server header to `<img src='x' onerror='alert(1)'/>` (as described in Section 4.3).

Remote code execution We use the XSS vulnerability described above to gain a foothold in the browser on the scanning host. For instance, we can inject a BeEF hook to remotely interact with the browser (as in Section 6.1). The hooked browser is the steppingstone to interact with the Metasploit Pro UI and trigger its controls. Interestingly, Metasploit Pro includes a *diagnostic console*, i.e., an embedded terminal that allows the analyst to run arbitrary commands on the underlying operating system.[8] Although the diagnostic console is disabled by default, the attacker can activate it through BeEF. In particular, the hooked browser is forced to perform a POST HTTP request to `/settings/update_profile` with the parameter `allow_console_access=1`. Since the diagnostic console is a browser-embedded Metasploit terminal emulator, the attacker can submit commands from the BeEF interface.

Takeover impact The Metasploit Pro documentation[9] clearly states that "Metasploit Pro Users Run as Root. If you log in to the Metasploit Pro Web UI, you can effectively run any command on the host machine as root". This opens a wide range of opportunities for the attacker. Among them, the most impactful is to establish a *reverse shell*. The reasons are twofold. First, opening a shell on the scanning host allows the attacker to execute commands directly on the operating system of the victim. Thus, attacks are no longer tunneled through the initial vulnerability, which might become unavailable, e.g., if Metasploit Pro is terminated. Second, a reverse shell works well even when certain network facilities, such as firewalls and NATs, are in place. Indeed, although these facilities may prevent incoming connections, usually they allow outgoing ones. Once a reverse shell is established, the attacker can access a permanent, privileged shell on the victim host.

6.3 Enhanced phishing

The goal of a phishing attack is to induce the victim to commit a dangerous action, e.g., clicking an untrusted URL or opening an attachment. In this section, we show how our attacker model changes phishing attacks, using CheckShortURL as an application scenario.

Traditional Phishing A common phishing scenario is that of an unsolicited email with a link pointing to a malicious web page, e.g., `http://ev.il`. The phishing site mimics a reputable, trusted web page. For instance, the attacker may clone a bank's web site so that unaware users submit their access credentials. Another technique is to provoke a reaction to an emotion, such as fear. This happens, for instance, with menacing alerts about imminent account locking and malware infections. Again, if victims believe that urgent action must be taken, they could overlook common precautions and, e.g., download dangerous files.

Defense mechanisms Most of the examples given above require the victim to open a phishing URL. Common, unskilled users typically evaluate the trustworthiness of a URL by applying their common sense.[10] Nevertheless, techniques such as URL shortening and open redirects [27] masquerade the phishing URL to resemble a trusted domain.

Some online services may help the user to detect phishing attacks. For instance, reputation systems and black/white lists, e.g., Web of Trust[11], can be queried for a suspect URL. However, phishing URLs often point to temporary websites that are unknown to these systems.

Since browsers automatically redirect without asking for confirmation, in [27] the authors highlight that victims can defend themselves by checking where the URL redirects without browsing it. To this aim, several online services, e.g., CheckShortURL, do redirect checking to establish the final destination of a redirect chain. Typically, the chain is printed in a report that the user inspects before deciding whether to proceed or not.

[7] https://help.rapid7.com/metasploit/release-notes/archive/2020/05/#20200514

[8] https://www.exploit-db.com/exploits/40415

[9] https://metasploit.help.rapid7.com/docs/metasploit-web-interface-overview

[10] E.g., see https://phishingquiz.withgoogle.com/

[11] https://www.mywot.com

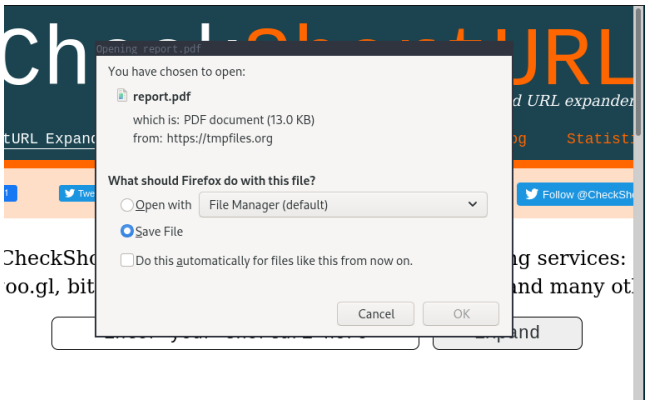

Figure 11: Phising through CheckShortURL.

Exploiting redirect checkers Redirect locations are contained in the Location header of the HTTP response asking for a redirection. According to our attacker model, this value is controller by the attacker. Thus, if the victim uses a vulnerable redirect checker, the report may convey an attack to the user browser. Since the goal is phishing, the attacker has two possibilities, i.e., forcing the URL redirection and exploit the scanning system reputation.

In the first case, the attacker delivers an XSS payload such as `window.location = "http://ev.il/"`. When it is executed, the browser is forced to open the given location and to redirect the user to the phishing site.

The second case is even more subtle. Since the XSS attack is delivered by the scanning system, the attacker can perform a phishing operation and ascribe it to the reporting system. For instance, the attacker can make the user browser download a malicious file pretending to be the scanning system pdf report. In this way, the attacker abuses the reputation of the scanning system to lure the victim. This can be achieved with the following payload.
`window.location="http://tmpfiles.org/report.pdf"`
The effect of injecting such a payload in CheckShortURL is shown in Figure 11.

7 Related Work

In this section, we survey the related literature.

7.1 Attacking the attacker

Although not frequent in the literature, the idea of attacking the attackers is not completely new. Its common interpretation is that the victim of an attack carries out a counter-strike against the host of the aggressor. However, even tracking an attack to its actual source is almost impossible if the attacker takes proper precautions (as discussed in Section 6.1). To the best of our knowledge, we are the first to consider the response-based exploitation of the attackers scanning systems.

Djanali et al. [9] define a low-interaction honeypot that simulates vulnerabilities to lure the attackers to open a malicious website. When this happens, the malicious website delivers a browser exploitation kit. The exploitation relies on a LikeJacking [29] attack to obtain information about the attacker's social media profile. Unlike our approach, their proposal substantially relies on social engineering and does not consider vulnerabilities in the attacker's equipment.

Also, Sintov [28] relies on a honeypot to implement a *reverse penetration* process. In particular, his honeypot attempts to collect data such as the IP address and the user agent of the attacker. Again, this proposal amounts to retaliating against the attackers after identifying them.

In terms of vulnerabilities, some researchers already reported weaknesses in scanning systems. The closest to our work is CVE-2019-5624 [4], a vulnerability in RubyZip that also affects Metasploit Pro. This vulnerability allows attackers to exploit *path traversal* to create a *cron job* that runs arbitrary code, e.g., to create a reverse shell. To achieve this, the attacker must import a malicious file in Metasploit Pro as a new project. However, as for [9], this attack requires social engineering as well as other conditions (e.g., about the OS used by the attacker). As far as we know, this is the only other RCE vulnerability reported for Metasploit Pro. Instead, apart from ours, no XSS vulnerabilities have been reported.

7.2 Security scanners assessment

Several authors considered the assessment of security scanners. However, they mainly focus on their effectiveness and efficiency in detecting vulnerabilities.

Doupé et al. [10] present WackoPicko, an intentionally vulnerable web application designed to benchmark the effectiveness of security scanners. The authors provide a comparison of how open source and commercial scanners perform on the different vulnerabilities contained in WackoPicko.

Holm et al. [18] perform a quantitative evaluation of the accuracy of security scanners in detecting vulnerabilities. Moreover, Holm [17] evaluated the performance of network security scanners, and the effectiveness of remediation guidelines.

Mburano et al. [23] compare the performance of OWASP ZAP and Arachni. Their tests are performed against the OWASP Benchmark Project [13] and the Web Application Vulnerability Security Evaluation Project (WAVSEP) [6]. Both these projects aim to evaluate the accuracy, coverage, and speed of vulnerability scanners.

To the best of our knowledge, there are no proposals about the security assessment of scanning systems. Among the papers listed above, none consider our attacker model or, in general, the existence of security vulnerabilities in security scanners.

7.3 Vulnerability detection

Many authors proposed techniques to detect software vulnerabilities. In principle, some of these proposals can be applied to scanning systems.

The general structure of vulnerability testing environments was defined by Kals et al. [19]. Our TEE implements their abstract framework by adapting it to inject responses instead of requests. The main difference is our test stub, that receives the requests from the scanning system under test. We substitute the crawling phase with tainted flow enumeration (see Section 4.2). During the attack phase, we substitute the payload list with a list of polyglots, which reduces testing time. Our exploit checker implements their analysis module as we also deal with XSS.

Many authors have proposed techniques to perform vulnerability detection through dynamic taint analysis. For instance, Xu et al. [32] propose an approach that dynamically monitors sensitive sinks in PHP code. It rewrites PHP source code, injecting functions that monitor data flows and detect injection attempts.

Avancini and Ceccato [3] also use dynamic taint analysis to carry out vulnerability detection in PHP applications. Briefly, they implement a testing methodology aiming at maximizing the code coverage. To check whether a certain piece of code was executed, they rewrite part of the application under test to deploy local checks.

These approaches rely on inspecting and manipulating the source code of the application under test. Instead, we work under a black-box assumption.

Besides vulnerability detection, some authors even use dynamic taint analysis to implement exploit detection and prevention methodologies. Vogt et al. [30] prevent XSS attacks by combining dynamic and static taint analysis in a hybrid approach. Similarly, Wang et al. [31] detect DOM-XSS attacks using dynamic taint analysis. Both these approaches identify sensitive data sinks in the application code and monitor whether untrusted, user-provided input reaches them.

Dynamic taint analysis techniques were also proposed for detecting vulnerabilities in binary code.

Newsome and Song [24] propose *TaintCheck*, a methodology that leverages dynamic taint analysis to find attacks in commodity software. *TaintCheck* tracks tainted sinks and detects when an attack reaches them. It requires a monitoring infrastructure to achieve this.

Clause et al. [7] propose a generic dynamic taint analysis framework. Similarly to [24], Clause et al. implement their technique for x86 binary executables. However, the theoretical framework could be adapted to fit our methodology.

In principle, the exploit prevention techniques mentioned above might be used to mitigate some of the vulnerabilities detected by RevOK. However, they do not deal with vulnerability detection. Moreover, they require access to the application code.

8 Conclusion

In this paper we introduced a new methodology, based on a novel attacker model, to detect vulnerabilities in scanning systems. We implemented our methodology and we applied our prototype RevOK to 78 real-world scanning systems. Our experiments resulted in the discovery of 36 new vulnerabilities. These results confirm the effectiveness of our methodology and the relevance of our attacker model.

Acknowledgements. This paper was partially funded by EU H2020 research project SPARTA (grant agreement n.830892).

References

[1] Wade Alcorn. Beef autorun rule engine. https://github.com/beefproject/beef/wiki/Autorun-Rule-Engine, Accessed March 19, 2020.

[2] Wade Alcorn. *The Browser Exploitation Framework*, Accessed March 3, 2020.

[3] Andrea Avancini and Mariano Ceccato. Towards Security Testing with Taint Analysis and Genetic Algorithms. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*, 2010.

[4] Luca Carettoni. On insecure zip handling, Rubyzip and Metasploit RCE (CVE-2019-5624). https://blog.doyensec.com/2019/04/24/rubyzip-bug.html, Accessed March 19, 2020.

[5] CheckShortURL. *CheckShortURL*, Accessed March 23, 2020.

[6] Shay Chen. The Web Application Vulnerability Scanner Evaluation Project. https://sourceforge.net/projects/wavsep/, Accessed March 19, 2020.

[7] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[8] MITRE Corporation. ATT&CK - Technical Information Gathering. https://attack.mitre.org/tactics/TA0015/, Accessed March 20, 2020.

[9] Supeno Djanali, FX Arunanto, Baskoro Adi Pratomo, Abdurrazak Baihaqi, Hudan Studiawan, and Ary Mazharuddin Shiddiqi. Aggressive web application honeypot for exposing attacker's identity. In *Proceedings of the 1st International Conference on Information Technology, Computer, and Electrical Engineering*, 2014.

[10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010.

[11] Ahmed Elsobky. Unleashing an Ultimate XSS Polyglot. https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot, Accessed March 19, 2020.

[12] OWASP Foundation. OWASP Top Ten. https://owasp.org/www-project-top-ten/, 2017.

[13] OWASP Foundation. OWASP Benchmark Project. https://owasp.org/www-project-benchmark/, Accessed March 19, 2020.

[14] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*, 2015.

[15] MUNSIRADO Group. *Nmap Online*, Accessed March 3, 2020.

[16] Web Hypertext Application Technology Working Group. *HTML Living Standard*, Last updated March 27, 2020.

[17] Hannes Holm. Performance of automated network vulnerability scanning at remediating security issues. *Computers & Security*, 2012.

[18] Hannes Holm, Teodor Sommestad, Jonas Almroth, and Mats Persson. A quantitative evaluation of vulnerability scanning. *Information Management & Computer Security*, 2011.

[19] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on World Wide Web*, 2006.

[20] Arturs Lavrenovs and F Jesús Rubio Melón. HTTP Security Headers Analysis of Top One Million Websites. In *Proceedings of the 10th International Conference on Cyber Conflict (CyCon)*, 2018.

[21] Arturs Lavrenovs and Gabor Visky. Investigating HTTP response headers for the classification of devices on the Internet. In *Proceedings of the 7th IEEE Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2019.

[22] Paul Leach, Michael Mealling, and Rich Salz. A universally unique identifier (UUID) urn namespace. 2005.

[23] Balume Mburano and Weisheng Si. Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark. In *Proceedings of the 26th International Conference on Systems Engineering (ICSEng)*, 2018.

[24] James Newsome, Dawn Song, James Newsome, and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.

[25] Nmap project. *Nmap*, Accessed March 23, 2020.

[26] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[27] Craig A Shue, Andrew J Kalafut, and Minaxi Gupta. Exploitable Redirects on the Web: Identification, Prevalence, and Defense. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.

[28] Alexey Sintsov. Honeypot that can bite: Reverse penetration. In *Black Hat Europe Conference*, 2013.

[29] SOPHOSLABS. Facebook worm: Likejacking. https://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/, Accessed on March 19, 2020.

[30] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[31] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *Journal of Parallel and Distributed Computing*, 2018.

[32] Wei Xu, Sandeep Bhatkar, and R Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. *Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook*, 2005.

A Vulnerability Disclosure

All the vulnerabilities reported in this paper were promptly notified to the scanning system vendors. We based our responsible disclosure process on the ISO 29147[12] guidelines. Below, we describe each disclosure step in detail and the vendors feedback.

A.1 First contact

The first step of our responsible disclosure process consisted of a non-technical email notification to each vendor. We report our email template below.

```
Dear <scanning system vendor>,

my name is <identification and links>

As part of my research activity on a
novel threat model, I found that your
platform is most likely vulnerable
to XSS attacks.
In particular, the vulnerability I
discovered might expose your end-users
to concrete risks.

For these reasons, I am contacting
you to start a responsible disclosure
process. In this respect, I am kindly
asking you to point me to the right
channel (e.g., an official bug bounty
program or a security officer to
contact).

Kind regards
```

We sent the email through official channels, e.g., contact mail or form, when available. For all the others, we tried with a list of 13 frequent email addresses, including security@, webmaster@, contact@, info@, admin@, support@.

In 5 cases the previous attempts failed. Thus, we submitted the corresponding vulnerabilities to OpenBugBounty.[13]

A.2 Technical disclosure

After the vendor answered our initial notification, providing us with the technical point of contact, we sent a technical report describing the vulnerability. The report was structured according to the following template, which was accompanied by a screenshot of the PoC exploit inside their system.

```
The issue is a Cross-Site Scripting
attack on your online vulnerability
scanning tool <scanning system name>.

This exposes your users to attacks,
possibly leading to data leakage and
account takeover.

A malicious server can answer with XSS
payloads instead of its standard headers.
For example, it could answer with this
(minimal) HTTP response:

<minimal PoC for the scanning system>

Since your website displays this data in
a report, this code displays a popup on
the user page, but an attacker can
include any JavaScript code in it,
taking control of the user browser (see
https://beefproject.com/), and hence
make them perform actions on your
website or steal personal information.

I attached a screenshot of the PoC
running on your page. The PoC is
completely harmless, both for your
website and for you to test.
I also hosted a malicious (but harmless)
server here if you want to reproduce the
issue: <test stub network address>

You can perform any scan you want
against it (please let me know if it is
offline).
```

In a few cases we extended the report with additional details, requested by some vendors. For example, some of them asked for the CVSSv3[14] calculation link and an impact evaluation specifically referring their scanning system.

A.3 Vendors feedback

Out of the 36 notifications, we received 12 responses to the first contact message. All the responses arrived within 2 days. Among the notified vendors 5 fixed the vulnerability within 10 days. Another vendor informed us that, although they patched their scanning system, they started a more general investigation of the vulnerability and our attacker model. This will result in a major update in the next future. Finally, after fixing the vulnerability, one of the vendors asked us not to appear in our research.

[12] https://www.iso.org/standard/72311.html
[13] https://www.openbugbounty.org
[14] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator