Writing Manual Shellcode by Hand

Overview

Assembly is the language to know when developing buffer overflow exploits in software exploitation scenarios. This short document is designed to give an introduction to this sort of programming language where the Windows API will be used to demonstrate how it is possible to call a message box directly by using hardcoded memory addresses.

Contents

Chapter 0 – Prerequisites 2

Chapter 1 – Preparing the Environment 3

Chapter 2 – Writing a Dummy Popup 5

Chapter 3 – Writing a "LOL" Popup 10

Chapter 4 – Using the Stack for a Popup 13

Chapter 5 – The Easy Way – Part 1 17

Chapter 6 – The Easy Way – Part 2 20

Ending Words & References 28

Chapter 0

Prerequisites

Tools

- Arwin (http://www.vividmachines.com/shellcode/arwin.c)
- Code::Blocks (http://www.codeblocks.org/)
- OllyDbg (http://www.ollydbg.de/version2.html)

Goals

- Create minimalistic shellcode
- Include custom text input

This is for educational purposes only.

Chapter 1

Preparing the Environment

First download and install the Code Blocks IDE environment, which you're going to use to compile the Arwin source code. Simply download "arwin.c" from the link in the previous chapter and open this in Code Blocks or your own preferred C compiler.

When you've opened this file in Code Blocks, browse to the "Build" menu and choose "Build".

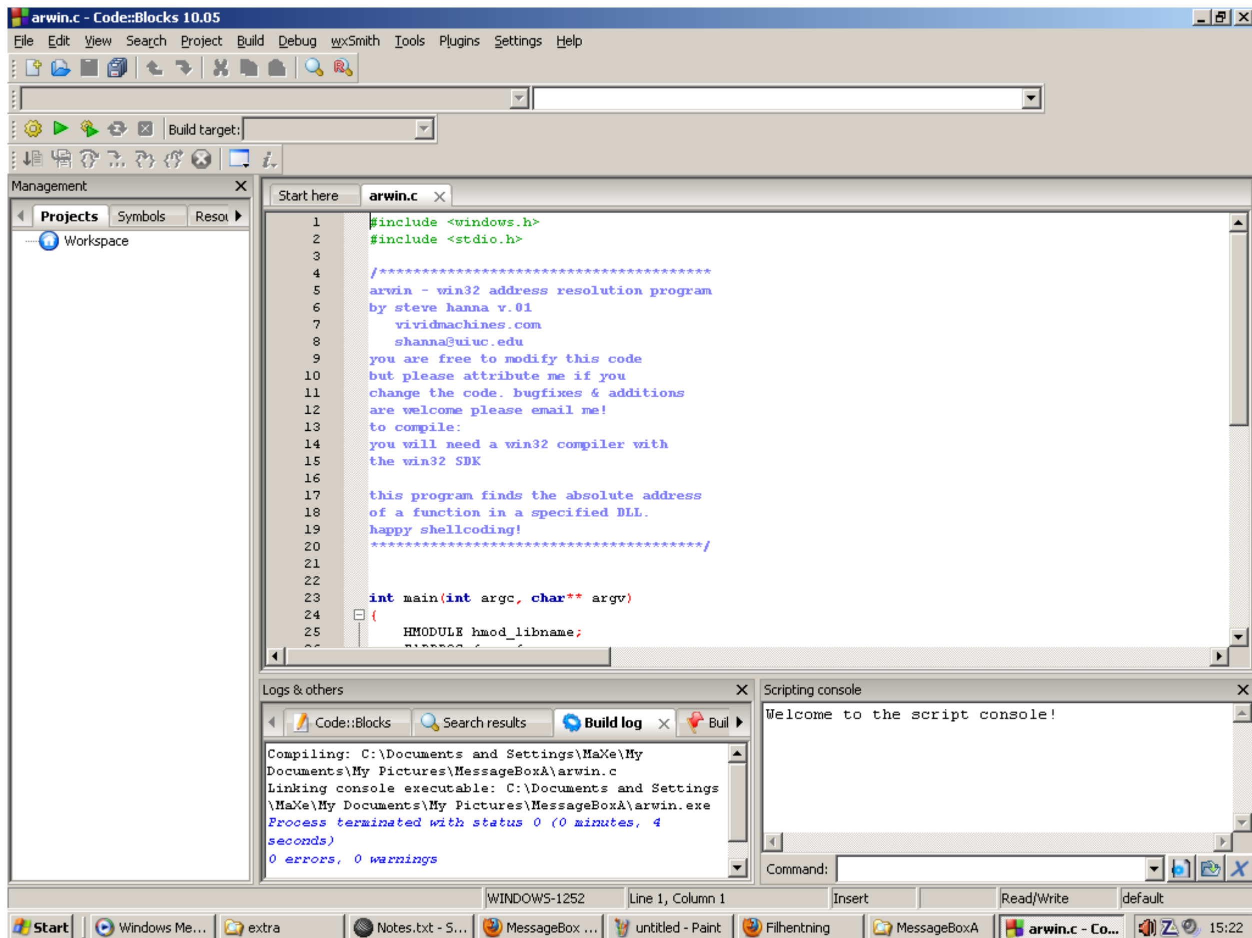

Figure 1.0.1 – Compiling Arwin.c in Code::Blocks

Now you're ready to use Arwin, which will be located in the same directory as where the source code is located. Browse to this location in a command prompt window. If the directory is located deep within your computer, consider copying arwin.exe to a directory near the root.

Then issue the following command without quotes: "arwin.exe User32.dll MessageBoxA".

When you've done this, you'll be searching for the MessageBoxA function within User32.dll

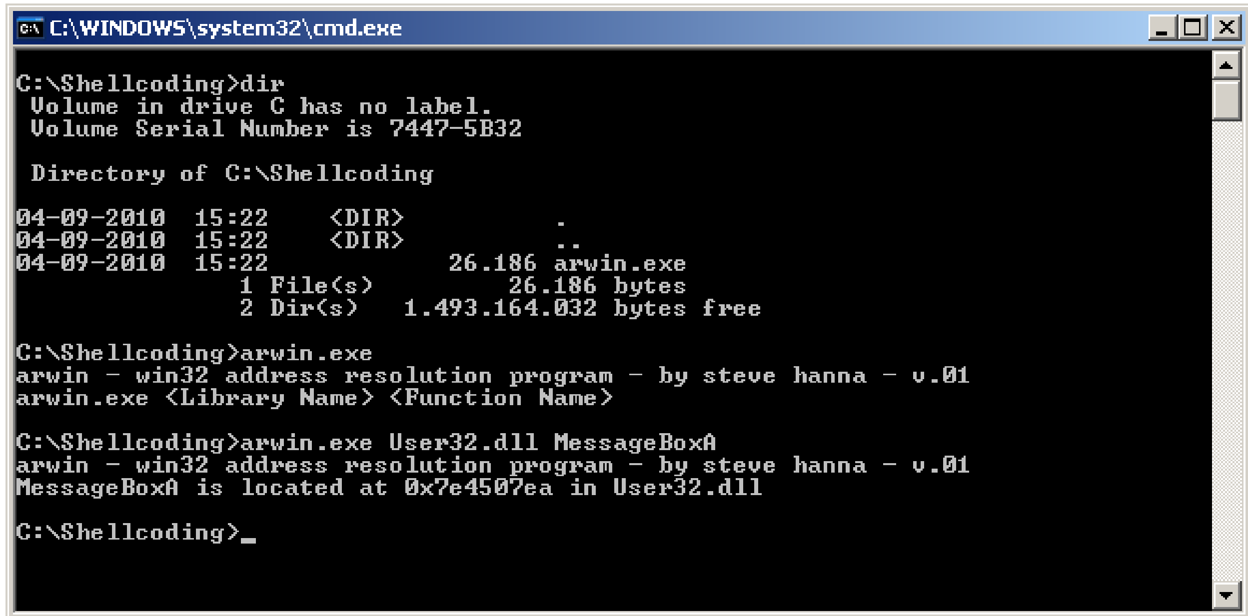

Figure 1.0.2 – Using Arwin.exe to find the MessageBoxA memory address

If you've done this right, you'll see a text string like: MessageBoxA is located at …

It is important to note, that this address is not static and it is therefore usually not the same on another machine. If you're running Windows Vista or 7, then the address will change each time you reboot your computer. If you're running certain IPS's on Windows XP, this will occur too.

After remembering these precautions, note down the memory address Arwin returned.

In our shellcode we're going to assume User32.dll is already loaded within the given executable which we are injecting our own shellcode into directly and manually. A file which already has this DLL file loaded when it is run is: GenuineCheck.exe

You can of course, use other files as well but I will be using this executable as an example.

(Any executable file with a graphical user interface should have User32.dll loaded.)

Chapter 2

Writing a Dummy Popup

Open OllyDbg and if you haven't downloaded this program yet, then download it and make sure you get version 2.0 which has a lot of improvements and new features. One of the new features is the ability to change the PE header directly, which we're going to do later on.

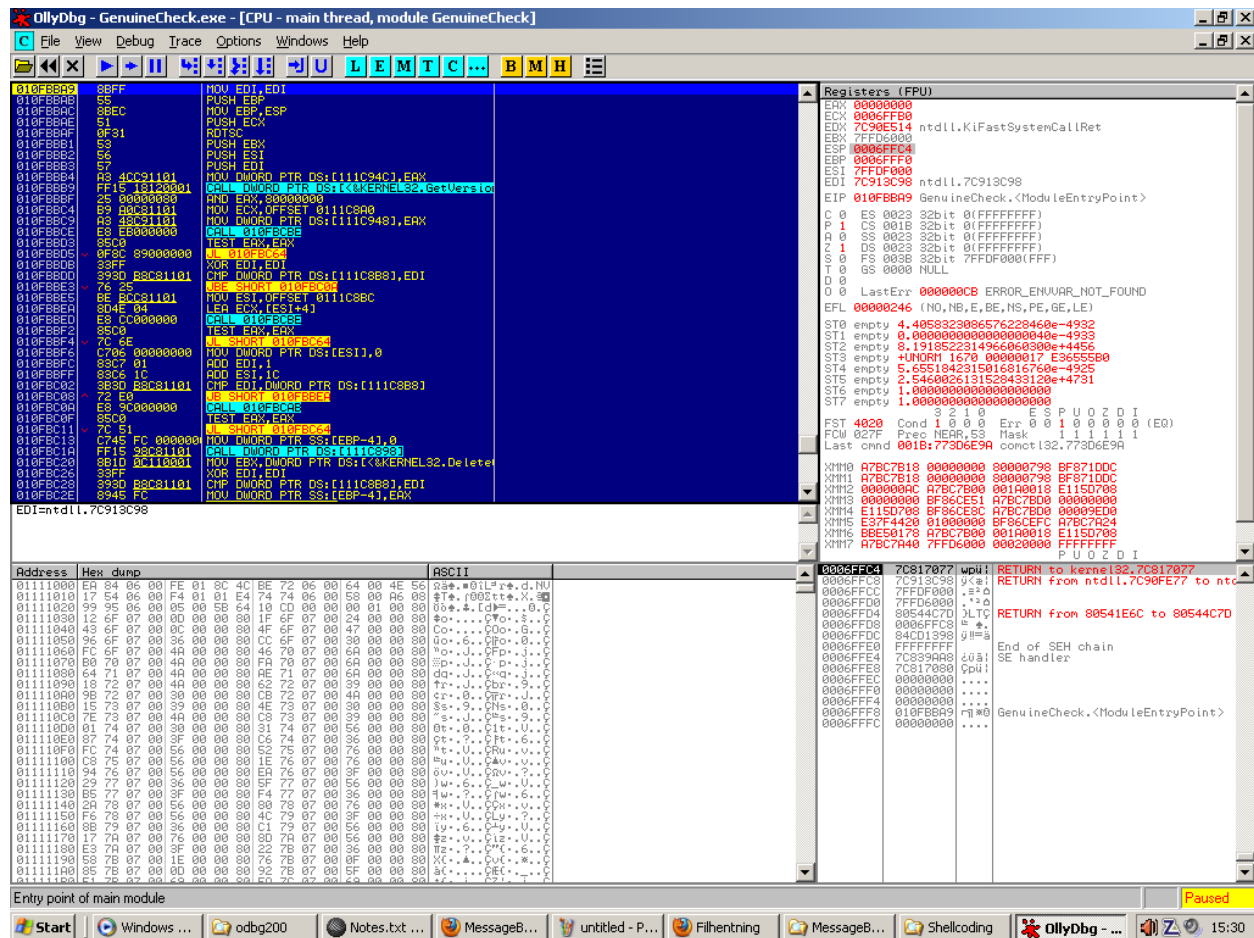

Figure 2.0.1 – An executable file within OllyDbg

Simply open your executable file as shown in Figure 2.0.1, and take a quick look at it. Don't try to understand what all of it means. In the lower right we have our stack, which stores values, variables, arguments, pointers, etc.

Essentially the same, hexadecimal values because binary values are harder to read.

In the lower left we have our "Hex Dump" window, which is useful when we're following perhaps a dynamic change of our shellcode, such as an encoding sub-routine.

The upper right is our registers and flags, while the upper left is the "disassembly window" where we can single-step aka trace through our assembly opcodes, aka (CPU) instructions.

Basically when we press F7 once, then one instruction is executed by your computer.

This may add or remove values from the stack, alter the registers, and of course if more opcodes aka instructions are used, then the hardware of the computer may be used along with perhaps API libraries to show pop boxes, connect to a remote attacker, or shut the computer down.

In essence the only limitation with Assembly is the hardware and your imagination.

Since we're not going to use the original program behind, change a good amount of the first couple of instructions visible to NOP's, so it is easier to see your own injected shellcode.

1. Select aka mark a region of opcodes shown in the "disassembly window".
2. Right click and browse to "Edit" then select: "Fill with NOPs".

All you should see is a lot of "NOP" instructions now. Press F7 a couple of times and see how you execute one NOP at a time. This opcode doesn't do anything, so don't worry if you execute many of them. As you may see, EIP changes each time you execute one NOP. This register that you can see on the right, points to the current instruction you're going to execute next.

F8 is used to jump over "CALL" opcodes but also single-step through our shellcode as well, while F9 executes the entire code until an event occurs, such as a popup box requiring attention.

Our initial dummy popup code looks like this:

```
MOV EAX, 0x7e4507ea
XOR EBX, EBX
PUSH EBX
PUSH EBX
PUSH EBX
PUSH EBX
CALL EAX
```

Figure 2.0.2 – Dummy Popup Shellcode

As you can see, I'm using this memory address: 0x7e4507ea to call the MessageBoxA function. Make sure not to use this address since it will most likely be different on your system. Use the address you found with the Arwin tool mentioned earlier.

The reason why we begin with a "dummy" is to make it easier to understand, how the call and assigned variables function together, inside your computer. Here is an explanation of the code:

1) Change EAX to 0x7e4507ea
2) XOR EBX with EBX. This alters EBX to 0. (zero)
3) Push the value of EBX to the stack.
4) Push the value of EBX to the stack.
5) Push the value of EBX to the stack.
6) Push the value of EBX to the stack.
7) Call the memory address which EAX is pointing to.

Note: In our case, we're calling MessageBoxA()

Why push the value of EBX 4 times to the stack? Because MessageBoxA takes 4 arguments!

Here's the MSDN syntax:

```
int WINAPI MessageBox(
  __in_opt  HWND hWnd,
  __in_opt  LPCTSTR lpText,
  __in_opt  LPCTSTR lpCaption,
  __in      UINT uType
);
```

Figure 2.0.3 – MessageBoxA syntax

Here's the full resource: http://msdn.microsoft.com/en-us/library/ms645505%28VS.85%29.aspx

I will explain what the arguments means later on and how to use them with Assembly code. But for now you should double-click a NOP instruction after or where the current Instruction Pointer (EIP) is pointing to. If you don't know where EIP is pointing to, look at the very left side in your debugger where the memory addresses are.

The memory address which is highlighted is the next instruction which will be executed when you hit and press F7. When you've double-clicked a NOP instruction, begin to write the code in yourself. Make sure to use your own memory address for MessageBoxA as described earlier.

When you've done this, press F7 once and see how EAX now contains your memory address which you used Arwin to find, if you press F7 again then EBX will become 0 and if you press F7 4 times more the stack will have 4 new values of 0 (zero) each.

Now you're at the "CALL EAX" instruction. If you want to, you can press F7 all the way through the function call, but if you just need to see your message box work or not, then press F8 and watch the beautiful yet very empty popup box.

If your popup looks like the one included in the screenshot below, then you've done right.

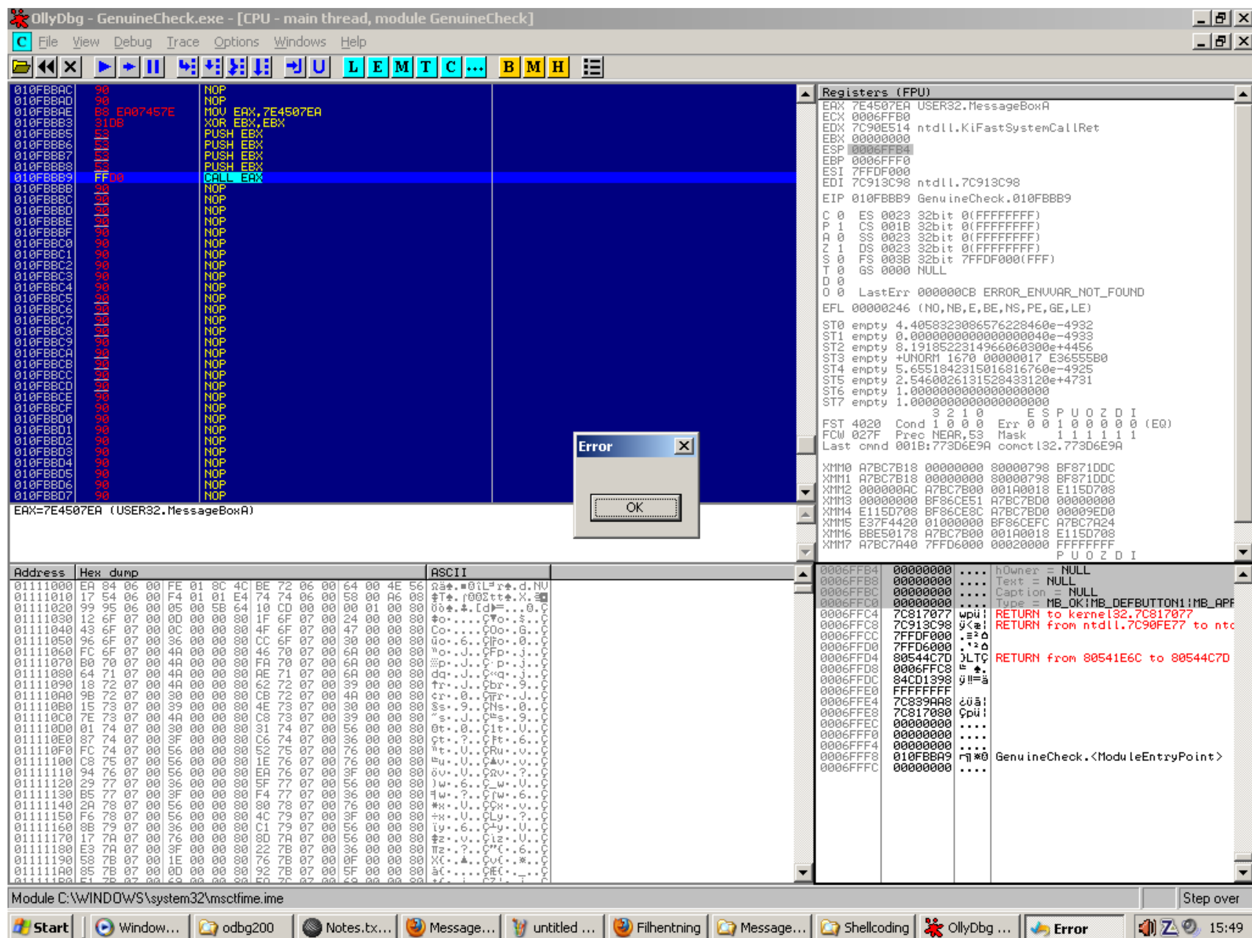

Figure 2.0.4 – A Dummy Popup working in OllyDbg

This is of course, not very impressive. Even a monkey can be taught how to do this, so we'll continue onwards and learn how to provide a custom text message now, which there are two ways which you can use in order to do this.

Before we do this, mark the shellcode you wrote. Then right-click it, browse to "Edit" and select "Binary copy" in order to save it. Paste the output into notepad and remove the spaces between.

This is the (hexadecimal) binary version of your recently created popup box. If you re-open the executable file you used by pressing CTRL+F2, overwrite the first couple of instructions with NOPs and then select a good amount of NOPs to overwrite with your binary code, then you can enter the "Edit" menu by right-clicking again and now you select "Binary paste".

Whenever you use "Binary paste", make sure to select **enough** opcodes!

If you don't, then only a part of your shellcode will be shown or some of the opcodes may be incomplete and therefore they may resemble something else which it shouldn't. Therefore you should as previously mentioned, always select enough opcodes aka instructions.

Before we continue I'd like to explain more about our dummy popup shellcode.

Q: Why is XOR used?

A: Because this avoids 0-bytes (00), which otherwise kills shellcode.

Q: What is the Stack used for in this scenario?

A: It's used to store arguments for MessageBoxA and possibly even custom text!

Q: I pressed F7 at "CALL EAX" and noticed a value was pushed onto the stack, why?

A: This value is the current EIP (Instruction Pointer) which is used to return back to the original code when a "RETN" opcode is executed. If this value wasn't pushed onto the stack, then the computer would return to whatever other value is stored on the stack, resulting in massive failure.

Q: I saw something like this on the stack, what does it mean?

```
0006FFB4   00000000  ....  ; hOwner = NULL
0006FFB8   00000000  ....  ; Text = NULL
0006FFBC   00000000  ....  ; Caption = NULL
0006FFC0   00000000  ....  ; Type = MB_OK|MB_DEFBUTTON1|MB_APPLMODAL
```

A: hOwner is the owner of the window, we'll use zero since we can. Text is a pointer to ASCII text stored somewhere in the memory, caption is the same as text and type means what kind of buttons (and more) that the popup box should display, do, require, etc.

An "ASCII Pointer" generally points to a memory address, containing a string in ASCII format with a 00 byte in the end to close the string. All this functionality will be explained in the next couple of chapters.

Chapter 3

Writing a “LOL” Popup

It’s time for a message box with at least some words in it, so we’ll start with 3 letters including a 0-byte due to a (32-bit) register can contain 4 bytes at once. We’ll use the phrase “LOL” without quotes, which equals “4C4F4C” in hexadecimal representation.

A quick note is that our arguments has to be pushed in reverse order, so the last argument we push to the stack is actually the first in the MessageBoxA (API) function call.

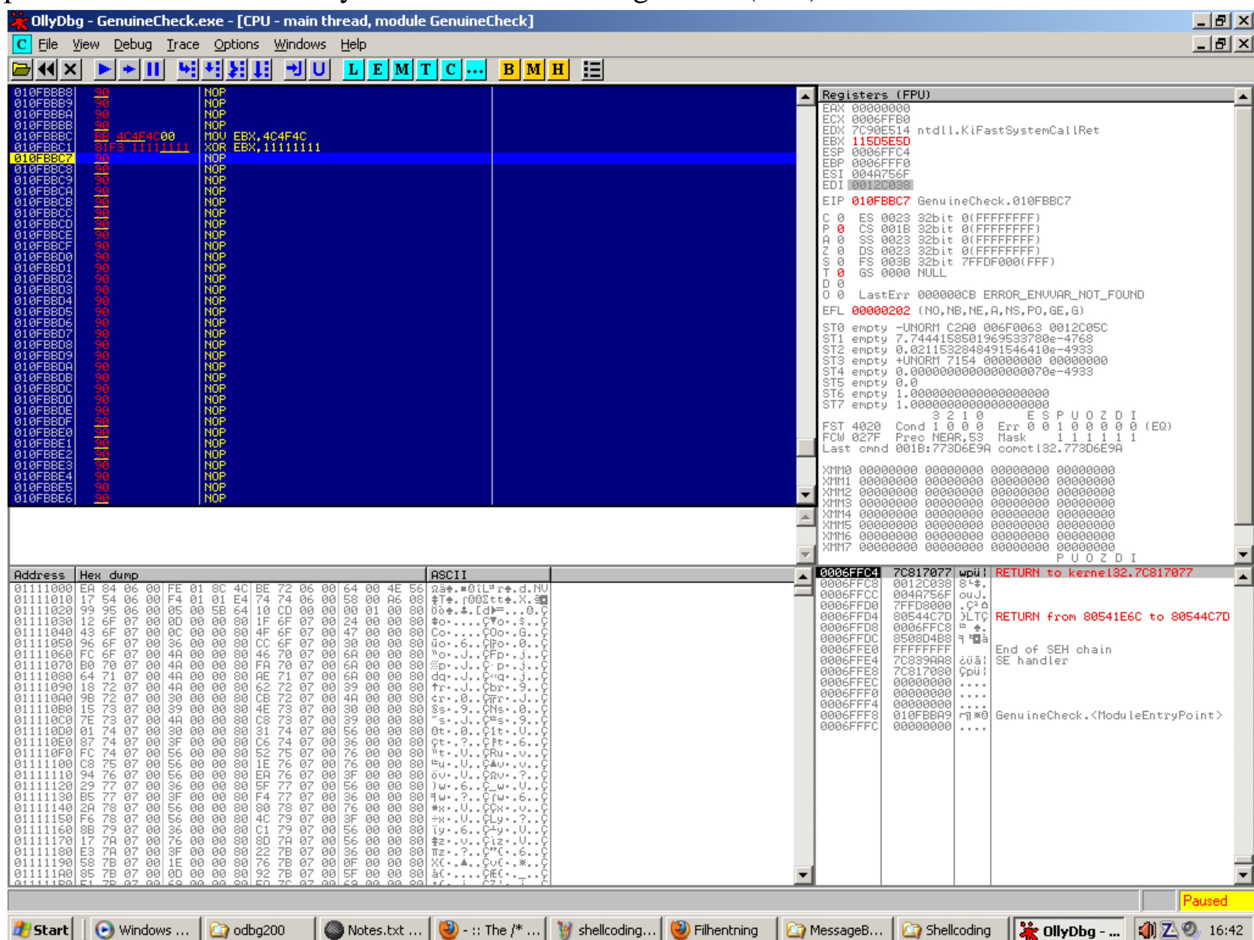

Figure 3.0.1 – Encoding “LOL” with XOR

First we're going to encode our string "LOL" by XOR'ing it. The complete string with a 0-byte looks like this: 004C4F4C. In Little Endian order it looks like this: 4C4F4C00, which is seen in figure 3.0.1 and afterwards XOR'd with 11111111 which alters it to: 115D5E5D

We're going to push this text string to the stack and use the stack pointer (ESP) as a reference to our string. It may sound a little bit confusing but in essence it is actually quite simple.

Figure 3.0.2 – Custom "LOL" Popup Box

The code used in the above figure is almost the same as our dummy popup code except that our custom string is included and that 2 more registers has been used. Very shortly described, our XOR'd string is inserted into EBX, then XOR'd with 1111 1111 which returns 004C4F4C.

Then EBX (our string) is pushed onto the stack and the current stack pointer is copied into EDI for future reference, because the stack pointer changes each time something is pushed onto the stack. Then ESI (0) is pushed to the stack, then EDI twice (our string) and then ESI.

Now EAX is called, and as you can see in the above figure, our arguments are correct.

You might want to try this out yourself, so copy the code below and adapt it to your needs.

```
Address   Hex dump          Command
010FBBB1    B8 EA07457E     MOV EAX,7E4507EA
010FBBB6    31F6            XOR ESI,ESI
010FBBB8    31FF            XOR EDI,EDI
010FBBBA    BB 5D5E5D11     MOV EBX,115D5E5D
010FBBBF    81F3 11111111   XOR EBX,11111111
010FBBC5    53              PUSH EBX
010FBBC6    89E7            MOV EDI,ESP
010FBBC8    56              PUSH ESI
010FBBC9    57              PUSH EDI
010FBBCA    57              PUSH EDI
010FBBCB    56              PUSH ESI
010FBBCC    FFD0            CALL EAX
```

Figure 3.0.3 – Custom “LOL” Popup Code

I should mention that you actually don’t need the XOR EDI, EDI opcode but I added it to make things a bit more clear. The only thing you need to change in the above code is MOV EAX to include the memory address for MessageBoxA which you found earlier using Arwin.

Do not try to copy these opcodes directly into OllyDbg, write them yourself. Pay attention to the opcode with our custom string, because if you write it wrong then our string may be displayed wrong or not at all. It’s very important you do things right, since there’s no margin for error.

When you’ve created your “LOL” popup box, do a binary copy which you can use later on:

```
B8 EA 07 45 7E 31 F6 31 FF BB 5D 5E 5D 11 81 F3
11 11 11 11 53 89 E7 56 57 57 56 FF D0
```

Figure 3.0.4 – Binary opcodes for a “LOL” popup

As you can see for yourself, we’ve used 29 bytes to create a message box saying “LOL”. It does not get any more efficient than this so make sure you understand it before continuing onto the next chapter which will include more advanced shellcode.

Chapter 4

Using the Stack for a Popup

After creating our popup with the words "LOL", we're ready to progress onto something a bit more advanced. The text string I'm going to use is: "Hello, this is MaXe from InterN0T.net".

In short we're going to push the string to the stack and then point to it. Easier said than done, but it is far from impossible. All it takes is time, and you got plenty of that to learn this. You might wonder why learn how to call a popup box? Well if you can do that, then you're able to learn by yourself how to use the rest of the API!

First we have to encode our string in hexadecimal, if you use The XSSOR, make sure to use "Machine Hex Encoding" and remove these "\x" so the string looks similar to the example.

```
48 65 6c 6c 6f 2c 20 74 68 69 73 20 69 73 20 4d 61 58 65 20 66 72 6f
6d 20 49 6e 74 65 72 4e 30 54 2e 6e 65 74
```

Figure 4.0.1 – Hexadecimal Encoding of our String

The above characters could be opcodes but they are in fact our string in its encoded form. Since we're going to push the string to the stack we need to do more work with it, yes this is painful but in the end it'll work and show the string as it should.

To begin with we add a 00-byte in the end of our string. Furthermore our string should be dividable by 4, if it isn't we add more 00-bytes. In our case after the first 00-byte we added, we notice that we need to add 2 more bytes to make our string more workable.

So there's 3x 00-bytes after the last byte (74), and to save ourselves trouble from doubling or tripling the work we have to do with the string due to the Little Endian Architecture, we read the 4 last bytes and put these in 1 row and then the next 4 bytes in the next row.

The first row with the 3x 00-bytes added looks like this: 00 00 00 74, the next row: 65 6E 2E 54.

Continue with this procedure until you reach the last 4 bytes and put them on the last row. The reason why we have to do this is because the last byte is read first on the IA-32 platform. Now it will probably become even more confusing later on, but if it works then you did it right.

To make it more clear, I've made a complete example below:

```
1.  00202074
2.  656E2E54
3.  304E7265
4.  746E4920
5.  6D6F7266
6.  20655861
7.  4D207369
8.  20736968
9.  74202C6F
10. 6C6C6548
```

Figure 4.0.2 – Our string in Little Endian format

As you can see, my first 4 bytes doesn't have 3x 00-bytes. I used "20" which equals a simple space, instead of two of the 00-bytes. I will use the example above, for the rest of this chapter.

The code and technique we're going to use is relatively the same as our "LOL" popup code, the only difference is that we have to do some XOR magic with the first line (1) in our string and then push all of it to the stack, and then use the stack pointer to point to it.

Figure 4.0.3 – Our string gets pushed to the stack

In the screenshot on the previous page, you're able to see that the string is pushed in yes, reverse order which is exactly how it should because this is how the Intel Architecture works. I know it's seems strange, but I promise there are more obscure things out there.

In essence row 2 to 10 was just pushed to the stack, while row 1 contained a 00 byte which had to be encoded which I did by XOR'ing the string with 1111 1111. Exactly the same trick I used in our "LOL" popup code, except that we're using a longer string this time and that it's only the first 4 bytes (actually the 4 last) that needs to be encoded.

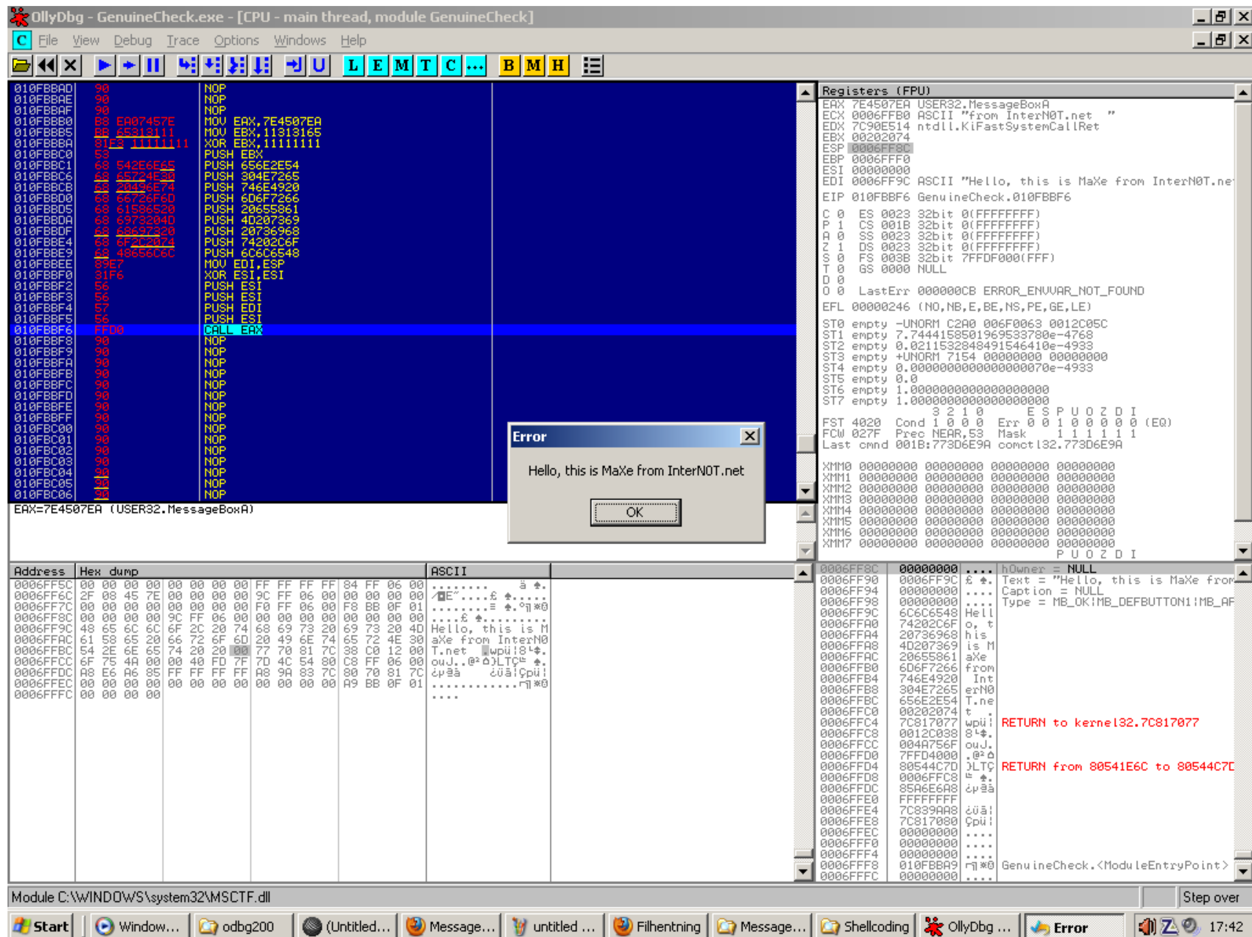

Figure 4.0.4 – Working custom popup box with no 00-bytes

When our string is pushed onto the stack, the stack pointer which points to our string is copied into the EDI register for future reference, then ESI (0) is pushed onto the stack twice. Then EDI is pushed which contains our ASCII (stack) pointer to our custom string, and finally ESI again (0) in order to complete the amount of necessary arguments.

Finally EAX is called and our popup box should be executed without any problems.

As per usual the code is available in the example below for you to use.

```
010FBBB0    B8 EA07457E      MOV EAX,7E4507EA
010FBBB5    BB 65313111      MOV EBX,11313165
010FBBBA    81F3 11111111    XOR EBX,11111111
010FBBC0    53               PUSH EBX
010FBBC1    68 542E6E65      PUSH 656E2E54
010FBBC6    68 65724E30      PUSH 304E7265
010FBBCB    68 20496E74      PUSH 746E4920
010FBBD0    68 66726F6D      PUSH 6D6F7266
010FBBD5    68 61586520      PUSH 20655861
010FBBDA    68 6973204D      PUSH 4D207369
010FBBDF    68 68697320      PUSH 20736968
010FBBE4    68 6F2C2074      PUSH 74202C6F
010FBBE9    68 48656C6C      PUSH 6C6C6548
010FBBEE    89E7             MOV EDI,ESP
010FBBF0    31F6             XOR ESI,ESI
010FBBF2    56               PUSH ESI
010FBBF3    56               PUSH ESI
010FBBF4    57               PUSH EDI
010FBBF5    56               PUSH ESI
010FBBF6    FFD0             CALL EAX
```

Figure 4.0.5 – Custom Popup Box Code

If you want to try this code right out of the box, on your own machine then simply edit the first instruction to the memory address pointing to MessageBoxA() on your system, and then enter each opcode manually into your debugger.

All of the push arguments are a bit tedious to push, so copy these one by one and insert them.

If you're really lazy, then you can use the binary codes below which are the push opcodes that pushes row 2 to 10 of our string. Remember, to select a large amount of opcodes in your debugger before your paste it in, and afterwards double-check all of the instructions indeed were pasted correctly into your debugger.

The fastest way to check this, is to check the first opcode is correct and that the last opcode is correct, and of course that the entire shellcode looks like the code in figure 4.0.5

```
68542E6E656865724E306820496E746866726F6D6861586520686973204D6868697320
686F2C20746848656C6C
```

Figure 4.0.6 – Binary Push Opcodes (Row 2 to 10)

Play a little with this and when you feel ready to continue onto the next chapter which is going to use another approach, then make sure you understood the previous parts because the next way is the hardest but also somewhat the most interesting.

Chapter 5

The Easy Way – Part 1

In this chapter we're going to simply add our string to the end of our shellcode. This means that we'll have a 00-byte in the end of our shellcode which may not work in real buffer overflow scenarios but in this test case it doesn't really matter. We'll solve that problem later on!

Now you may remember that we need to point to our string somehow, and that this pointer uses memory addresses. Just copy "EIP" into another register, is what you may think. This isn't possible so we actually have to calculate the EIP, no kidding.

There are many ways to Rome and my method is just one of them. It's made of pure logic and how the Assembly opcode "CALL" functions. This opcode pushes the current instruction pointer when executed and then it jumps to the memory address or register defined.

We can use this to our advantage by taking this value from the stack and storing it in a register, but we have to be careful because we can't just take a value from the stack without putting it back, since this may cause unexpected errors and more in real shellcode. (I.e. backdoors)

Take a look at this sub-routine I made: (Pseudo Code)

```
PUSH
---- SOI ---- (Start of Instructions)
POP EDI
PUSH EDI
RETN
NOP
---- EOI ---- (End of Instructions)
CALL ESP
```

Figure 5.0.1 – Calculating EIP Sub-routine

In short: Push the binary values between SOI and EOI to the stack, and then call the stack which jumps directly to the stack, take the EIP from the stack into EDI, push the same value back and then return to where the last pushed value on the stack is pointing to.

This may seem a bit weird, but now EDI contains EIP, and we need that!

So how do we find out the binary values of POP EDI, PUSH EDI, RETN and NOP? Quite simple, we write these in our debugger one by one but don't execute them. This results in these binary opcodes: 5F 57 C3 90

Now don't forget that these values needs to be in Little Endian order, so reverse them till they look like this: 90 C3 57 5F and then implement them in your PUSH opcode, and write it in your debugger like this: PUSH 90C3575F

We also need to create some dummy code to make things a lot easier. The code we're going to use almost looks like all the previous examples we've used, where we use ESI for our null (0) values and EDI for our ASCII pointer which refers to our custom text string.

```
010FBBB0    68 5F57C390     PUSH 90C3575F
010FBBB5    FFD4            CALL ESP
010FBBB7    90              NOP
010FBBB8    31F6            XOR ESI,ESI
010FBBBA    B8 EA07457E     MOV EAX,7E4507EA
010FBBBF    66:83C7 01      ADD DI,1
010FBBC3    90              NOP
010FBBC4    56              PUSH ESI
010FBBC5    56              PUSH ESI
010FBBC6    57              PUSH EDI
010FBBC7    56              PUSH ESI
010FBBC8    90              NOP
010FBBC9    FFD0            CALL EAX
```

Figure 5.0.2 – Dummy Popup Code

Now this code does not include our text string yet, but when it's done then we need to place it after the last CALL EAX opcode. I've added a few NOP's in this code to separate the different parts so it is easier to read and understand.

The first 2 opcodes calculates EIP and when the first NOP is hit, EDI contains the value of EIP.

This changes though as we continue to execute our code, so after we've written all these opcodes manually into our debugger we need to adjust "ADD DI, 1" to add the amount of bytes there is from CALL ESP till our text string. In short the space from the first NOP and till the beginning of our text string. This equals 21 in decimal and in hexadecimal, 15.

This is very important to keep in mind that you're working with hexadecimal and not ordinary decimal values. If you forget this like I occasionally do, your shellcode will fail at some point.

Now our text string needs to be encoded into binary (hexadecimal) code before pasting it into our debugger. The string I'm going to use is: "Did you know InterN0T.net is the best community?".

When we've encoded it by e.g. using The XSSOR, removed \x from the string and added a 00-byte it looks like the example below.

```
44696420796f75206b6e6f7720496e7465724e30542e6e657420697320746865
206265737420636f6d6d756e6974793f00
```

Figure 5.0.3 – Our string encoded in binary form

You may wonder why I did not reverse it into Little Endian? Well we don't need to do that when we're using our text string this way. This saves us some trouble, but there's still the 00-byte at the end which we'll have to deal with later on. But for now let us just see if it works.

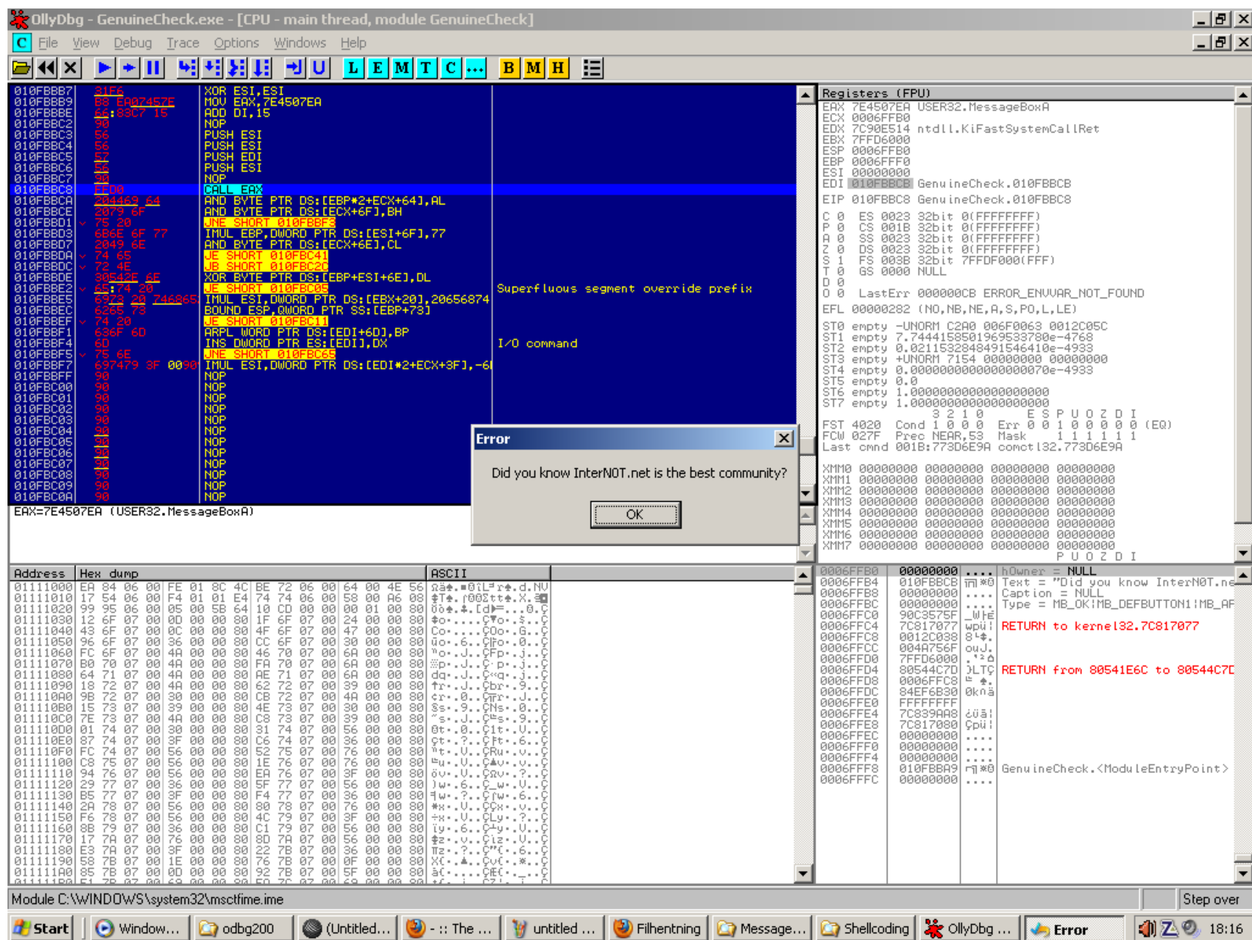

Figure 5.0.4 – Working custom messagebox

It works as you can see. Don't worry about the "opcodes" after CALL EAX, since we're going to assume that we won't execute these afterwards. If we wanted to make sure this wouldn't happen we could jump somewhere else in memory or perform a system exit call.

There's no need to worry about that, since you can learn this later on if it's required.

Chapter 6

The Easy Way – Part 2

We got our custom message box working in the last chapter but we would like to eliminate the 00-byte which could cause our shellcode to be terminated in a real buffer overflow scenario.

In order to do this we have to either encode that part of our shellcode or encode all of it! This is both fun and interesting to do even though it is also a bit strange at first. We're going to encode a part of our shellcode by "XOR'ing" it, which I have also explained in my other paper about bypassing anti-virus scanners.

In this case we need to use another approach since we have to assume we do not know any hard memory addresses, which means we have to either calculate the start and the end of our string, search for the start and the end and then encode that or what I find the most easy, encode the amount of bytes there is in our shellcode.

First we need to know how big our previous shellcode is:

```
68 5F 57 C3 90 FF D4 90 31 F6 B8 EA 07 45 7E 66
83 C7 15 90 56 56 57 56 90 FF D0 20 44 69 64 20
79 6F 75 20 6B 6E 6F 77 20 49 6E 74 65 72 4E 30
54 2E 6E 65 74 20 69 73 20 74 68 65 20 62 65 73
74 20 63 6F 6D 6D 75 6E 69 74 79 3F 00
```

Figure 6.0.1 – Binary Custom Popup Box

The above binary code is exactly 77 bytes long aka 4D in hexadecimal. This is very useful as you'll see when we implement our custom XOR encoder which will also function as a decoder, making things easier for us.

In essence our encoder (and decoder) will have to find out the memory address of the first byte to encode and then loop through X amount of bytes which will be XOR'd. These amounts of bytes are equal to how big our custom popup box is.

In order to find the first memory address of the first byte to XOR encode, we're going to use our previous method used to calculate EIP and then add a value to it and thereby adjusting it.

Now before I go ahead and explain this, a typical problem with XOR encoding in the “section” of the program you’re writing your custom opcodes to, is that it’s not writable by the program.

If it isn’t writable you’ll get an Access Violation Error, and the “XOR” opcode won’t work.

Therefore open the Memory Map in OllyDbg by pressing ALT+M or the “M” icon. Find the name of the executable file you’re injecting your code into, and look for “PE Header”.

Then double-click this and scroll down to where you see “.text” mentioned again.

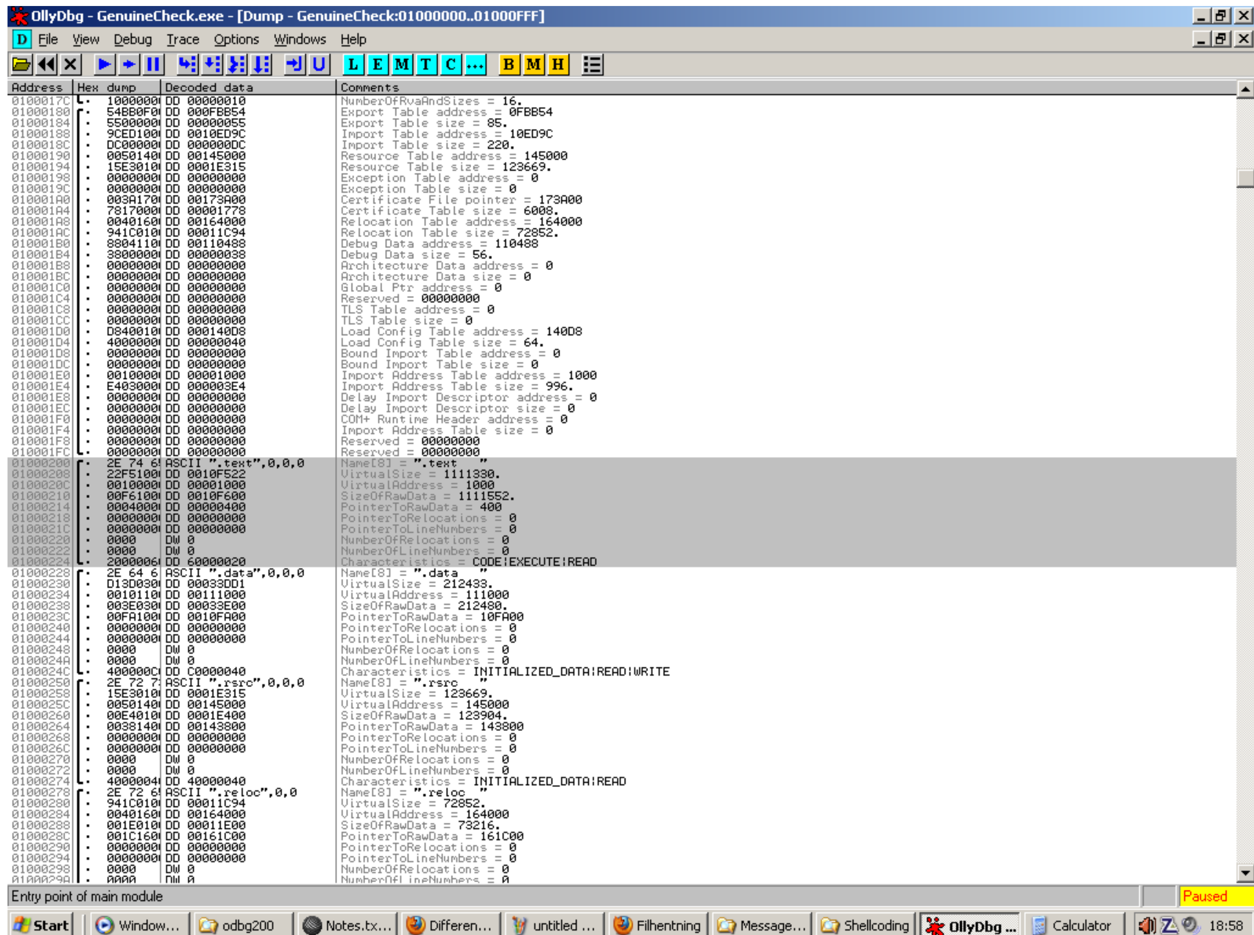

Figure 6.0.2 – PE Header of our executable file

Now double click the “Characteristics” line and edit only the first hexadecimal character to E.

This will make the .text section executable, readable and most importantly: Writeable.

If you want to know more about these sections and how they function, then I suggest you read my paper about bypassing anti-virus scanners but also read up on PE files since this should explain everything you may want to know about this topic.

Now in order to use this change, we need to save these changes and open the new file.

Do this by right clicking the marked and changed line, browse to "Edit" and select: "Copy to Executable". When you're in this new window, right click and choose "Save file". Now open the newly created file which you hopefully saved under a new filename.

After we've done this we're ready to write our custom encoder with the following code.

```
010FBBA9      68 5F57C390    PUSH 90C3575F
010FBBAE      FFD4           CALL ESP
010FBBB0      66:83C7 10     ADD DI,10
010FBBB4      31C9           XOR ECX,ECX
010FBBB6      80C1 4D        ADD CL,4D
010FBBB9      8037 0D        XOR BYTE PTR DS:[EDI],0D
010FBBBC      47             INC EDI
010FBBBD      E2 FA          LOOP SHORT 010FBBB9
```

Figure 6.0.3 – Custom XOR Encoder

You might remember the first line, this calculates the EIP and at the third line where EDI contain the current EIP, we add 16! Yes that's six-teen, not 10 because we're working with hexadecimal numbers and not decimal numbers. This is very important to keep in mind.

Then ECX is zeroed out (nulled) so we can use this to count down from 77 bytes. ADD CL, 4D adds 77 to ECX and the reason why we use CL and not ECX is to avoid 00 bytes in our shellcode since ADD ECX takes 4 bytes as input while CL takes 1 byte.

XOR BYTE PTR DS:[EDI],0D. Now that may be giving you headache but it's actually quite simple. Another explanation of this instruction could be: XOR the Byte with 0D, which EDI is pointing to. This will change the byte to another value, which is good to avoid Anti-Virus scanners and also in order to avoid 00 bytes.

When this XOR instruction is executed the first time, EDI is pointing to the byte right after the LOOP opcode. After XOR has been executed, EDI is increased by 1 so it points to the next byte. Then a jump is performed back to "XOR" and 1 is deducted from ECX.

This procedure continues until ECX is equal to 0. When ECX is 0, the "LOOP Jump" is not taken and therefore the rest of the shellcode is executed.

Now this encoder will mess up our popup box completely, but it'll encode it and avoid the 00-byte which is our goal. When this code has been encoded, we can copy this new and much obfuscated code, and when that is run it automatically decodes itself without any changes made.

It isn't that efficient in this case, but if our payload containing a lot of 00-bytes it would be very efficient in my opinion since it's hard to write any smaller encoder than that.

First write the encoder yourself in the debugger, and if your shellcode is longer or shorter than 77 bytes then make 100% sure that you have calculated the right value for the "ADD CL" opcode.

Furthermore, when you write "LOOP SHORT" in your debugger, do NOT use the memory address I used. Look to the left of your debugger at where the "XOR" opcode is, then note the memory address and write that in exactly as it is.

If you want to check that you have written the right memory address, select the LOOP opcode only and you should see a long arrow pointing to the XOR instruction. If it doesn't then you've written something wrong and then you need to write the right memory address of course.

After this, paste your binary popup box shellcode after the LOOP SHORT opcode.

Figure 6.0.4 – Custom XOR Encoder (1)

When you execute the XOR instruction the first time, and you see the first byte turn into something else, then you've written the right value for the "ADD DI, X" instruction.

Now to get a clear overview of you encoding your shellcode, right click EDI and select “Follow in dump”. This allows you to see the shellcode change each time you pass the XOR instruction.

Do NOT hold F7 too long or you will “crash” into your encoded shellcode which may alter it in a way so you can’t use it and then you’ll have to redo it all again. If you however want to encode your shellcode the quick way, then make sure you have executed XOR once.

Then select the opcode right after LOOP and press F2.

This sets a breakpoint and if you press F9 then all of your shellcode will be encoded instantly, do not press any more buttons because you don’t need to do that for now.

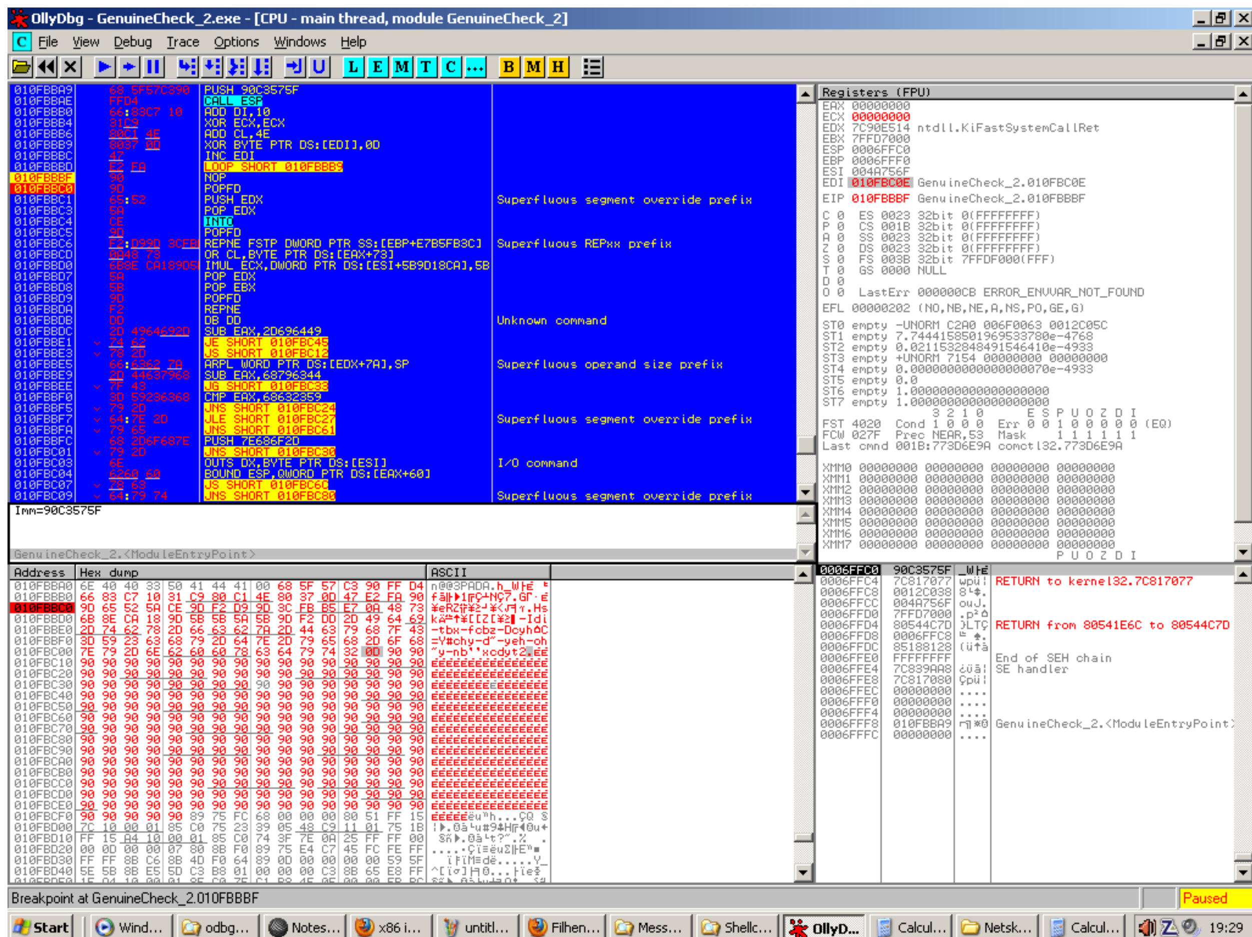

Figure 6.0.5 – Custom XOR Encoder (2)

If the last part of your shellcode ends with 0D, then all of your shellcode has been encoded correctly. Now select all your shellcode, including the encoder / decoder and do a binary copy.

Make sure to get everything from the beginning to the end. If the last obfuscated (encoded) instruction makes it impossible to copy the last byte of your shellcode alone, copy this too and then manually remove the unnecessary bytes (hopefully NOPs) afterwards in your text editor.

In my code example I had to remove a new NOP's (90) at the end.

When and if you've done everything correct, then you'll see your popup box like below.

Figure 6.0.6 – Custom XOR Encoder (decoder now) Working

For good old times you can get the binary code below.

```
685F57C390FFD46683C71031C980C14E80370D47E2FA909D65525ACE9DF2D99D
3CFBB5E70A48736B8ECA189D5B5B5A5B9DF2DD2D4964692D7462782D6663627A
2D446379687F433D59236368792D647E2D7965682D6F687E792D6E6260607863
647974320D
```

Figure 6.0.7 – Binary Code of Custom Encoder Popup Box

The entire un-encoded shellcode looks like this:

```
Address   Hex dump         Command
Comments
010FBBA9     68 5F57C390   PUSH 90C3575F
010FBBAE     FFD4          CALL ESP
010FBBB0     66:83C7 10    ADD DI,10
010FBBB4     31C9          XOR ECX,ECX
010FBBB6     80C1 4D       ADD CL,4D
010FBBB9     8037 0D       XOR BYTE PTR DS:[EDI],0D
010FBBBC     47            INC EDI
010FBBBD     E2 FA         LOOP SHORT 010FBBB9
010FBBBF     90            NOP
010FBBC0     90            NOP
010FBBC1     68 5F57C390   PUSH 90C3575F
010FBBC6     FFD4          CALL ESP
010FBBC8     90            NOP
010FBBC9     31F6          XOR ESI,ESI
010FBBCB     B8 EA07457E   MOV EAX,7E4507EA
010FBBD0     66:83C7 15    ADD DI,15
010FBBD4     90            NOP
010FBBD5     56            PUSH ESI
010FBBD6     56            PUSH ESI
010FBBD7     57            PUSH EDI
010FBBD8     56            PUSH ESI
010FBBD9     90            NOP
010FBBDA     FFD0          CALL EAX
010FBBDC     204469 64     AND BYTE PTR DS:[EBP*2+ECX+64],AL
010FBBE0     2079 6F       AND BYTE PTR DS:[ECX+6F],BH
010FBBE3     75 20         JNE SHORT 010FBC05
010FBBE5     6B6E 6F 77    IMUL EBP,DWORD PTR DS:[ESI+6F],77
010FBBE9     2049 6E       AND BYTE PTR DS:[ECX+6E],CL
010FBBEC     74 65         JE SHORT 010FBC53
010FBBEE     72 4E         JB SHORT 010FBC3E
010FBBF0     30542E 6E     XOR BYTE PTR DS:[EBP+ESI+6E],DL
010FBBF4     65:74 20      JE SHORT 010FBC17
010FBBF7     6973 20 74686 IMUL ESI,DWORD PTR DS:[EBX+20],20656874
010FBBFE     6265 73       BOUND ESP,QWORD PTR SS:[EBP+73]
010FBC01     74 20         JE SHORT 010FBC23
010FBC03     636F 6D       ARPL WORD PTR DS:[EDI+6D],BP
010FBC06     6D            INS DWORD PTR ES:[EDI],DX
010FBC07     75 6E         JNE SHORT 010FBC77
010FBC09     697479 3F 009 IMUL ESI,DWORD PTR DS:[EDI*2+ECX+3F],-6F
```

Figure 6.0.8 – Un-Encoded Shellcode

While the encoded shellcode looks like this:

```
Address   Hex dump          Command
Comments
010FBBA9      68 5F57C390   PUSH 90C3575F
010FBBAE      FFD4          CALL ESP
010FBBB0      66:83C7 10    ADD DI,10
010FBBB4      31C9          XOR ECX,ECX
010FBBB6      80C1 4E       ADD CL,4E
010FBBB9      8037 0D       XOR BYTE PTR DS:[EDI],0D
010FBBBC      47            INC EDI
010FBBBD      E2 FA         LOOP SHORT 010FBBB9
010FBBBF      90            NOP
010FBBC0      9D            POPFD
010FBBC1      65:52         PUSH EDX
010FBBC3      5A            POP EDX
010FBBC4      CE            INTO
010FBBC5      9D            POPFD
010FBBC6      F2:D99D 3CFBB REPNE FSTP DWORD PTR SS:[EBP+E7B5FB3C]
010FBBCD      0A48 73       OR CL,BYTE PTR DS:[EAX+73]
010FBBD0      6B8E CA189D5B IMUL ECX,DWORD PTR DS:[ESI+5B9D18CA],5B
010FBBD7      5A            POP EDX
010FBBD8      5B            POP EBX
010FBBD9      9D            POPFD
010FBBDA      F2            REPNE
010FBBDB      DD            DB DD
010FBBDC      2D 4964692D   SUB EAX,2D696449
010FBBE1      74 62         JE SHORT 010FBC45
010FBBE3      78 2D         JS SHORT 010FBC12
010FBBE5      66:6362 7A    ARPL WORD PTR DS:[EDX+7A],SP
010FBBE9      2D 44637968   SUB EAX,68796344
010FBBEE      7F 43         JG SHORT 010FBC33
010FBBF0      3D 59236368   CMP EAX,68632359
010FBBF5      79 2D         JNS SHORT 010FBC24
010FBBF7      64:7E 2D      JLE SHORT 010FBC27
010FBBFA      79 65         JNS SHORT 010FBC61
010FBBFC      68 2D6F687E   PUSH 7E686F2D
010FBC01      79 2D         JNS SHORT 010FBC30
010FBC03      6E            OUTS DX,BYTE PTR DS:[ESI]
010FBC04      6260 60       BOUND ESP,QWORD PTR DS:[EAX+60]
010FBC07      78 63         JS SHORT 010FBC6C
010FBC09      64:79 74      JNS SHORT 010FBC80
010FBC0C      320D 90909090 XOR CL,BYTE PTR DS:[90909090]
```

Figure 6.0.9 – Encoded Shellcode

Ending Words

Knowing Assembly in order to write and create shellcode directly is indeed a very good idea since it allows the ethical hacker to create efficient, minimalistic and optimized shellcode for future exploitation scenarios which may require hardcore expertise.

Therefore you should if you don't already know Assembly, want to learn more. Don't go hard-core but try to create your own shellcode which calls a socket and listens for connections, or perhaps execute a system command (calc.exe?), or maybe another function from the API.

If you play long enough with the language it won't be as confusing, most of the time.

References

[1] http://www.intern0t.net
[2] http://www.ollydbg.de/
[3] http://www.uninformed.org/?v=5&a=3&t=pdf
[4] http://www.offensive-security.com