

R4ndom's Tutorial #22: Code Caves

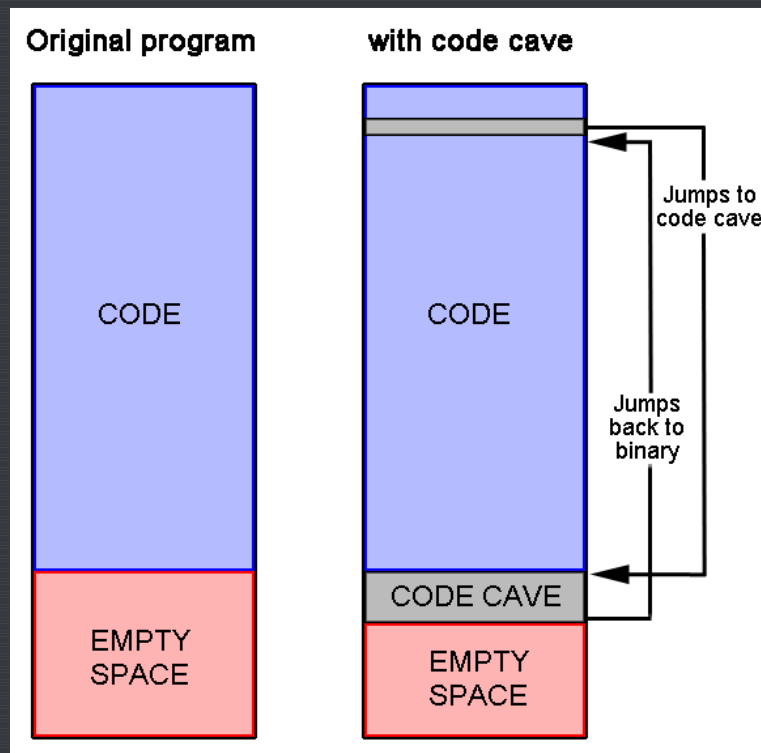
by R4ndom on Sep.20, 2012, under Intermediate, Reverse Engineering, Tutorials

In this tutorial we will be talking about code caves as well as PE sections, and touching on the PE header. We will be adding code caves to two crackmes, both available in the download of this tutorial. We will also be using the Multimate Assembler plugin which is also available in the download, as well as LordPE and CFF Explorer which are available on the [tools](#) page. This tutorial, as well as all of my others, can be downloaded on the [tutorials](#) page.

Introduction

Code caves are a way of adding our own code to a compiled binary. There are several reasons we may want to add our own code to a binary: we may want to add functionality (just see any of my tutorials on modifying binaries), we may want to change the way a program works by having it run our code instead of (or in addition to) the binary's own code, or we may wish to make a form of keygenner, as we will in this tutorial.

Here's how it works: First, we find an area in the binary that is not being used. This will be our 'cave'. We then insert our own code in this empty space, using Olly to assemble the actual instructions. Finally, we add a jump to our code cave somewhere in the original binary so that our code gets executed. At the end of the code cave, we then return execution back to the original binary:



Sometimes the target binary does not have enough free space to fit our added code. In that case, we can add a completely new section to the binary, set it to execute privileges, and then jump to this new section. We will be doing this in the second half of this tutorial.

First, we will use a code cave to make a keygenner of sorts. What I mean is, we are going to add code to

Double-clicking on the badboy brings us to that area of the code:

00401284	INC EBX	kernel32.BaseThreadInItThunk
00401285	DEC EAX	
00401286	JNZ SHORT crackme.00401278	
00401288	PUSH crackme.00406749	String2 = ""
0040128D	PUSH crackme.00406549	String1 = crackme.00406549
00401292	CALL <JMP.&kernel32.lstrcpyA>	lstrcpyA
00401297	PUSH 200	Count = 200 (512.)
0040129C	PUSH crackme.00406949	Buffer = crackme.00406949
004012A1	PUSH 64	ControlID = 64 (100.)
004012A3	PUSH DWORD PTR SS:[EBP+8]	hWnd = 7FFD4000
004012A6	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
004012AB	PUSH crackme.00406549	String2 = ""
004012B0	PUSH crackme.00406949	String1 = ""
004012B5	CALL <JMP.&kernel32.lstrcmpA>	lstrcmpA
004012BA	OR EAX,EAX	kernel32.BaseThreadInItThunk
004012BC	JNZ SHORT crackme.004012D4	
004012BE	PUSH 40	Style = MB_OK MB_ICONASTERISK MB_APPLMODAL
004012C0	PUSH crackme.004062D8	Title = "Good boy..."
004012C5	PUSH crackme.004062AC	Text = "Yep, thats the right code!\n\nrGo write
004012CA	PUSH FF75 08	hOwner = 7FFD4000
004012CD	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004012D2	JMP SHORT crackme.004012E8	
004012D4	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
004012D6	PUSH crackme.00406306	Title = "Bad boy..."
004012DB	PUSH crackme.004062E7	Text = "Nope, thats not it!\n\nrTry again"
004012E0	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004012E3	PUSH 200	Length = 200 (512.)
004012E8	PUSH crackme.00406549	Destination = crackme.00406549
004012ED	CALL <JMP.&kernel32.RtlZeroMemory>	RtlZeroMemory
004012F2	PUSH 200	Length = 200 (512.)
004012F7	PUSH crackme.00406349	Destination = crackme.00406349
004012FC	CALL <JMP.&kernel32.RtlZeroMemory>	RtlZeroMemory
00401301	PUSH 200	Length = 200 (512.)
00401306	PUSH crackme.00406749	Destination = crackme.00406749
0040130B	CALL <JMP.&kernel32.RtlZeroMemory>	RtlZeroMemory
00401310	MOV ECX,16	
00401315	LEA ESI,DWORD PTR DS:[406311]	kernel32.7615ED6C
0040131A	LEA EDI,DWORD PTR DS:[406327]	kernel32.7615ED6C
00401320	REP MOVS BYTE PTR ES:[EDI],BYTE PTR	
00401326	POP EBX	
00401328	POP EAX	
00401329	JMP SHORT crackme.00401364	
0040132A	CMF CM,5FF	

Clicking on the first line of the badboy code, we can see that patching this app would be very easy:

004012B5	CALL <JMP.&kernel32.lstrcmpA>	lstrcmpA
004012BA	OR EAX,EAX	kernel32.BaseThreadInItThunk
004012BC	JNZ SHORT crackme.004012D4	
004012BE	PUSH 40	Style = MB_OK MB_ICONASTERISK MB_APPLMODAL
004012C0	PUSH crackme.004062D8	Title = "Good boy..."
004012C5	PUSH crackme.004062AC	Text = "Yep, thats the right code!\n\nrGo write
004012CA	PUSH FF75 08	hOwner = 7FFD4000
004012CD	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004012D2	JMP SHORT crackme.004012E8	
004012D4	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
004012D6	PUSH crackme.00406306	Title = "Bad boy..."

Of course, we're not patching this app. What we want to do is change the target so that instead of a badboy, it shows us the proper registration code.

Scrolling up the binary, we see that the main decryption/conversion code starts at address 40112C:

00401126	JNZ crackme.0040132C	
0040112C	PUSH EAX	Beginning of routine
0040112D	PUSH EBX	
0040112E	PUSH EBP	
0040112F	PUSH 200	Count = 200 (512.)
00401134	PUSH crackme.00406349	Buffer = crackme.00406349
00401139	PUSH 3EA	ControlID = 3EA (1002.)
0040113E	PUSH DWORD PTR SS:[EBP+8]	hWnd = 7FFD4000
00401141	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
00401146	CMP EAX,3	
00401149	JA SHORT crackme.00401163	
0040114B	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
0040114D	PUSH crackme.00406306	Title = "Bad boy..."
00401152	PUSH crackme.0040620A	Text = "Username must have at least 4 chars..
00401157	PUSH DWORD PTR SS:[EBP+8]	hOwner = 7FFD4000
0040115A	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
0040115F	LEAVE	
00401165	RET 10	
00401168	LEA EDX,DWORD PTR DS:[406349]	String = "h"
0040116D	PUSH EDX	lstrlenA
00401172	CALL <JMP.&kernel32.lstrlenA>	kernel32.BaseThreadInItThunk
00401175	MOV EBP,EAX	
0040117A	MOV ECX,5	
0040117E	XOR ESI,ESI	
00401183	XOR EAX,EAX	
00401188	MOV CL,BYTE PTR DS:[ESI+EDX]	kernel32.BaseThreadInItThunk
0040118D	MOV BL,CL	
00401192	MOV BL,BYTE PTR DS:[EAX+406328]	kernel32.BaseThreadInItThunk
00401195	INC EAX	
00401198	CMP EAX,5	
0040119B	MOV BYTE PTR DS:[EDX+ESI],BL	
0040119E	MOV BYTE PTR DS:[EAX+406327],CL	
004011A3	JNZ SHORT crackme.00401196	kernel32.BaseThreadInItThunk
004011A6	XOR EAX,EAX	
004011AB	INC ESI	
004011AE	CMP ESI,EBP	
004011B3	JB SHORT crackme.0040117A	
004011B6	XOR EDI,EDI	
004011BB	XOR ECX,ECX	
004011C0	TEST EBP,EBP	
004011C3	JB SHORT crackme.004011C9	
004011C6	MOV BL,BYTE PTR DS:[EDI+40632D]	
004011CB	MOV ESI,EBP	
004011CE	SUB ESI,ECX	
004011D3	DEC ESI	
004011D6	MOV AL,BYTE PTR DS:[EDX+ESI]	
004011DB	XOR BL,AL	
004011E0	INC EDI	
004011E3	MOV BYTE PTR DS:[ESI+ESI],BL	

It first get's our entered username, then checks to make sure it's at least 4 characters long. After this, the code begins to convert the username into a registration code. At the end of this, the converted registration code will be compared to our entered code to see if they match:

0040127E	88B8 49674000	MOV BYTE PTR DS:[EBX+4067491],CL	kernel32.BaseThreadInitThunk
00401284	43	INC EBX	String2 = ""
00401285	48	DEC EAX	String1 = crackme.00406749
00401286	75 F0	JNZ SHORT crackme.00401278	lstrcpyA
00401288	68 49674000	PUSH crackme.00406749	Count = 200 (512.)
0040128D	68 49654000	PUSH crackme.00406549	Buffer = crackme.00406949
00401292	E8 5F010000	CALL <JMP.&kernel32.lstrcpyA>	ControlID = 3EA (1002.)
00401297	68 00020000	PUSH 200	hWnd = 7FFD4000
0040129C	68 49694000	PUSH crackme.00406949	GetDlgItemTextA
004012A1	6A 64	PUSH 64	String2 = ""
004012A3	FF75 08	JMP SHORT crackme.00401283	String1 = ""
004012A6	68 55000000	PUSH 55000000	lstrcmpA
004012AB	68 49654000	PUSH crackme.00406549	kernel32.BaseThreadInitThunk
004012B0	68 49694000	PUSH crackme.00406949	Style = MB_OK MB_ICONASTERISK MB_APPLMODAL
004012B5	E8 36010000	CALL <JMP.&kernel32.lstrcmpA>	Title = "Good boy..."
004012BA	0BC0	OR EAX,EAX	Text = "Yep, thats the right code!\n\rGo write
004012BC	75 16	JNZ SHORT crackme.004012D4	hOwner = 7FFD4000
004012BE	6A 40	PUSH 40	MessageBoxA
004012C0	68 DB624000	PUSH crackme.0040620B	Style = MB_OK MB_ICONHAND MB_APPLMODAL
004012C5	68 AC624000	PUSH crackme.004062AC	Title = "Bad boy..."
004012CA	FF75 08	JMP SHORT PTR SS:[EBP+8]	Text = "Nope, thats not it!\n\rTry again"
004012CD	E8 CA000000	CALL <JMP.&user32.MessageBoxA>	hOwner = 7FFD4000
004012D2	EB 10	JMP SHORT crackme.004012E8	MessageBoxA
004012D4	6A 10	PUSH 10	Count = 200 (512.)
004012D6	68 06634000	PUSH crackme.00406306	Buffer = crackme.00406349
004012DB	68 E7624000	PUSH crackme.004062E7	ControlID = 3EA (1002.)
004012E0	FF75 08	JMP SHORT PTR SS:[EBP+8]	hWnd = 00320392 ('LaFarge's crackme #2',
004012E3	E8 B4000000	CALL <JMP.&user32.MessageBoxA>	GetDlgItemTextA
004012E8	EB 10	JMP SHORT crackme.00401278	String2 = ""

GetDlgItemTextA get's the entered code and lstrcmpA will compare it to the code that has been converted from our username. If they match, we fall through to the goodboy. If not, we jump to the badboy.

This means, at address 4012AB, the converted serial is pushed on to the stack to be compared with the entered code, pushed at the next address, 4012B0. We know that the entered code is pushed at 4012B0 because we can see that in the GetDlgItemTextA call, 406949 is the buffer where the dialog item was saved, and at address 4012B0, we see that this same buffer was pushed in the call to lstrcmpA. Because of this, we know that 406549, the other address that is pushed in the lstrcmpA function call, is the address for the buffer that the converted code is in.

*** One thing that came up in running this crackme in Olly was access violations and exceptions that the target would not handle. To overcome this, you can either try running the target over again and it usually works, or you can temporarily remove the OllyAdvanced plugin from the plugins folder- we won't be using that plugin in this tutorial anyway. ***

Let's try stepping through the code to see if we're right. Set a breakpoint at address 40112C (the beginning of the conversion routine) and run the app. Enter "R4ndom" in the username and anything you want in the reg. code. I entered "12121212". Now click the "Check it!" button and Olly will break:

0040111C	0F85 5B020000	JNZ crackme.0040137D	
00401122	6613D EC03	CMP AX,3EC	
00401126	0F85 00020000	JNZ crackme.0040132C	
0040112B	50	PUSH EAX	Count = 200 (512.)
0040112D	53	PUSH EBX	Buffer = crackme.00406349
0040112E	55	PUSH EBP	ControlID = 3EA (1002.)
0040112F	68 00020000	PUSH 200	hWnd = 00320392 ('LaFarge's crackme #2',
00401134	68 49634000	PUSH crackme.00406349	GetDlgItemTextA
00401139	68 EA030000	PUSH 3EA	String2 = ""
0040113E	FF75 08	JMP SHORT PTR SS:[EBP+8]	String1 = ""
00401141	E8 4A020000	CALL <JMP.&user32.GetDlgItemTextA>	lstrcmpA
00401146	83F8 03	CMP EAX,3	kernel32.BaseThreadInitThunk
00401149	77 18	JA SHORT crackme.00401163	Style = MB_OK MB_ICONHAND MB_APPLMODAL
0040114B	6A 10	PUSH 10	Title = "Bad boy..."
0040114D	68 06634000	PUSH crackme.00406306	Text = "Username must have at least 4 ch
00401152	68 0A624000	PUSH crackme.0040620A	hOwner = 00320392 ('LaFarge's crackme #2
00401157	FF75 08	JMP SHORT PTR SS:[EBP+8]	MessageBoxA
0040115A	E8 3D020000	CALL <JMP.&user32.MessageBoxA>	Count = 200 (512.)
0040115F	C9	LEAVE	Buffer = crackme.00406349
00401160	C2 1000	RETN 10	ControlID = 3EA (1002.)
00401163	8D15 49634000	LEA EDI,DWORD PTR DS:[406349]	hWnd = 00320392 ('LaFarge's crackme #2
00401169	52	PUSH EDI	GetDlgItemTextA
0040116A	E8 8D020000	CALL <JMP.&kernel32.lstrlenA>	String2 = ""
0040116F	8BE8	MOV EBP,EAX	String1 = ""
00401171	EB 10	JMP SHORT crackme.0040112B	lstrlenA

Single step until you get to address 401141. This is the instruction where the crackme is going to get our entered username. In the arguments passed to GetDlgItemTextA you can see that one of them is labeled 'buffer'. This is the buffer that the dialog item that is retrieved will go:

00401122	0F85 5B020000	JNZ crackme.0040137D	
00401126	6613D EC03	CMP AX,3EC	
0040112B	50	PUSH EAX	Count = 200 (512.)
0040112D	53	PUSH EBX	Buffer = crackme.00406349
0040112E	55	PUSH EBP	ControlID = 3EA (1002.)
0040112F	68 00020000	PUSH 200	hWnd = 00320392 ('LaFarge's crackme #2
00401134	68 49634000	PUSH crackme.00406349	GetDlgItemTextA
00401139	68 EA030000	PUSH 3EA	String2 = ""
0040113E	FF75 08	JMP SHORT PTR SS:[EBP+8]	String1 = ""
00401141	E8 4A020000	CALL <JMP.&user32.GetDlgItemTextA>	lstrcmpA
00401146	83F8 03	CMP EAX,3	kernel32.BaseThreadInitThunk
00401149	77 18	JA SHORT crackme.00401163	Style = MB_OK MB_ICONHAND MB_APPLMODAL
0040114B	6A 10	PUSH 10	Title = "Bad boy..."
0040114D	68 06634000	PUSH crackme.00406306	Text = "Username must have at least 4 ch
00401152	68 0A624000	PUSH crackme.0040620A	hOwner = 00320392 ('LaFarge's crackme #2
00401157	FF75 08	JMP SHORT PTR SS:[EBP+8]	MessageBoxA
0040115A	E8 3D020000	CALL <JMP.&user32.MessageBoxA>	Count = 200 (512.)
0040115F	C9	LEAVE	Buffer = crackme.00406349
00401160	C2 1000	RETN 10	ControlID = 3EA (1002.)
00401163	8D15 49634000	LEA EDI,DWORD PTR DS:[406349]	hWnd = 00320392 ('LaFarge's crackme #2
00401169	52	PUSH EDI	GetDlgItemTextA
0040116A	E8 8D020000	CALL <JMP.&kernel32.lstrlenA>	String2 = ""
0040116F	8BE8	MOV EBP,EAX	String1 = ""

We can see this happen by right-clicking on address 401134 (where the buffer address is pushed) and selecting "Follow in dump"->"Immediate Constant". This will load the dump window beginning at address 406349, which we can see is filled with zeroes. Now step once over the GetDlgItemTextA call and you will see our username show up in the dump:

Address	Hex dump	ASCII
00406349	52 34 6E 64 6F 6D 00 00 00 00 00 00 00 00 00 00	R4ndom.....
00406359	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00406369	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00406379	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00406389	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00406399	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004063A9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004063B9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004063C9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
004063D9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

So now we know that our username will be kept at address 406349. Let's let Olly know for future reference. Right-click the first byte in the dump window (address 406349) and choose 'Label'. Type in "Username buffer" and click OK. Now, anytime this address is used in Olly, we will see our custom label:

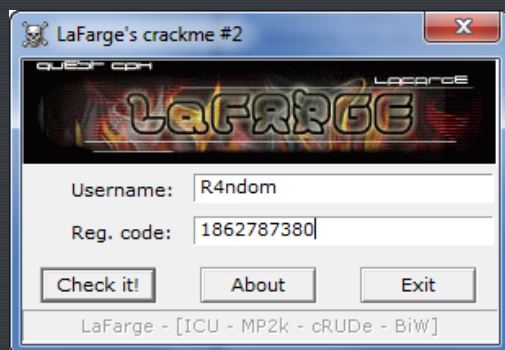
Continue stepping and we will first jump over the message that the username must be over 4 characters. We then call `IstrlenA`. We don't really need to know this but the encryption algorithm uses this value as part of the encryption process.

We then start stepping over the main encryption algorithm. You will see our username get transformed (several times). You can load the address in the dump if you would like to see them get modified in real time. We don't really care about this algorithm in this tutorial as we are not going to use it. If we were making a keygen, this part would be our most important code.

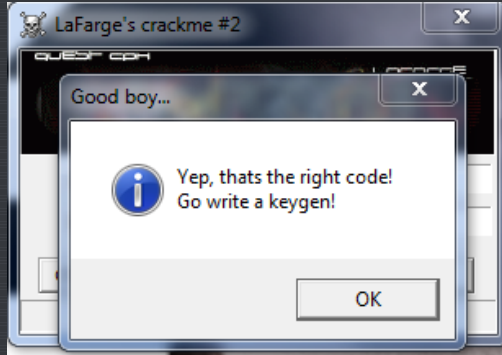
Go ahead and step till address 401288 (or place a BP here and run the target). This section is where we compare the converted username against the serial we entered. The first thing the crackme does is to copy the converted string to a new location to compare it. We can see both the correct serial, as well as a new buffer address pushed as arguments to the `IstrcpyA` function:

We then get the entered serial (with `GetDlgItemTextA`) and compare the two with `IstrcmpA`.

Now we can see if we're right by restarting the target, entering our original username (R4ndom) and entering the correct serial we see in the disassembly for the serial to see if it works:



and clicking "Check it!":



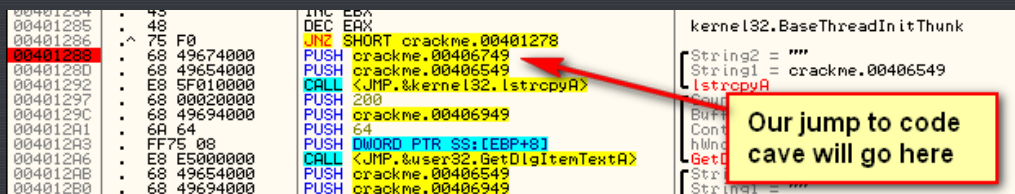
we see we are on the right track.

Now, we want to begin planning our code cave.

Starting the Code Cave

The first thing we have to do is decide where we will call our cave from. We must remember that wherever we put our jump (or call) will overwrite code, so we must either overwrite code we don't mind losing or we need to copy the instructions we are going to overwrite and paste them at the end of the code cave, thereby running the overwritten instructions right before we return.

Since we aren't actually going to check the entered serial with the real serial, we could overwrite the `IstrcpyA` call at address 401288. Another option is the `IstrcmpA` call at 4012B5. Frankly, we don't really need anything after the correct serial is computed except the message box that's going to display it. So let's go ahead and use the space where the first `IstrcpyA` is at address 401288:



Next we have to find a suitable place to put our own code into, our 'code cave'. Because the code section (as well as all other sections) has a minimum size a single block of code can be, unless the last block is exactly filled up with code, the remaining space will be filled with zeroes. If the code goes one byte over this block limit, an entirely new block will be created. The minimum a block of a section can be is stored in the PE header and is called *SectionAlignment*. This is set by default to 1000h, so if our code takes 1 byte or 1000-1 bytes, it will fit in the block. At one more byte, the operating system will allocate an additional block of 1000h bytes. If this happens, we will have our last block of code contain one byte of code and an additional section of FFFh zeroes. This is where we normally stick our code cave.

Normally, in a small program, we can simply scroll until we find this long series of zeroes and put it there. Other times we are not so lucky. Occasionally, there is not enough space to fit our code into. When this happens, we must create a new section ourselves, something we will do in the second part of this tutorial.

Some may wonder if we could put our cave in another section already in the target, such as `.data`, `.rsrc`, or the `.idata` sections. Theoretically you could, but the problem arises that these sections are not set up to execute code. Each section in a binary has a certain set of characteristics set up for read-only, write, executable etc. The `.code` section is usually the only section with the 'executable' flag set. We could put our cave in another section, but we would have to manually set this bit to allow code to be executable in it.

Fortunately, in this first part, scrolling down several pages we see we have plenty of room:

00404192	FF25	78504000	JMP	DWORD	PTR	DS:[&winmm.waveOutReset	winmm.waveOutReset
00404198	FF25	90504000	JMP	DWORD	PTR	DS:[&winmm.waveOutRestart	winmm.waveOutRestart
0040419E	FF25	94504000	JMP	DWORD	PTR	DS:[&winmm.waveOutUnpre	winmm.waveOutUnprepareHeade
004041A4	FF25	98504000	JMP	DWORD	PTR	DS:[&winmm.waveOutWrite	winmm.waveOutWrite
004041AA	00	00	DB	00			
004041B0	00	00	DB	00			
004041B8	00	00	DB	00			
004041C0	00	00	DB	00			
004041C8	00	00	DB	00			
004041D0	00	00	DB	00			
004041D8	00	00	DB	00			
004041E0	00	00	DB	00			
004041E8	00	00	DB	00			
004041F0	00	00	DB	00			
004041F8	00	00	DB	00			

Start of empty code section

So we will put our code cave starting at address 4041B0 (I usually try to put a buffer between the end of code and the beginning of the cave, just in case I need to change something):

00404198	FF25	90504000	JMP	DWORD	PTR	DS:[&winmm.waveOutReset	w
0040419E	FF25	94504000	JMP	DWORD	PTR	DS:[&winmm.waveOutUnpre	w
004041A4	FF25	98504000	JMP	DWORD	PTR	DS:[&winmm.waveOutWrite	w
004041AA	00	00	DB	00			
004041B0	00	00	DB	00			
004041B8	00	00	DB	00			
004041C0	00	00	DB	00			
004041C8	00	00	DB	00			
004041D0	00	00	DB	00			
004041D8	00	00	DB	00			
004041E0	00	00	DB	00			
004041E8	00	00	DB	00			
004041F0	00	00	DB	00			
004041F8	00	00	DB	00			

Our code cave will start here

Now we need to think about what our code cave is going to do. First, it will load the correct (converted) serial into a register, or rather the *address* of this serial. It will then push this address onto the stack, along with a couple of additional values that the message box requires. We will then jump back to the original code directly to the call to MessageBoxA. This allows us to use the targets code for calling the MessageBox function. It also allows the target to continue going after the message box is displayed. As far as the target is concerned, it has just shown us the badboy, so after we click OK it will simply continue as if we pressed the OK button on the badboy.

In order to see how we need to set up the variables for our call to MessageBoxA, we can simply look at the target and see how it was done originally:

004012BE	6A 40	PUSH 40	Style = MB_OK MB_ICONASTERISK MB_APPLMODAL
004012C0	68 D8624000	PUSH crackme.0040620B	Title = "Good boy..."
004012C5	68 AC624000	PUSH crackme.004062AC	Text = "Yep, thats the right code!\n\nGo write
004012CA	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner = 7FFD6000
004012D0	E8 CA000000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004012D2	EB 14	JMP SHORT crackme.004012E8	
004012D4	6A 10	PUSH 10	Style = MB_OK MB_ICONHAND MB_APPLMODAL
004012D6	68 06634000	PUSH crackme.00406306	Title = "Bad boy..."
004012D8	68 E7624000	PUSH crackme.004062E7	Text = "Nope, thats not it!\n\nTry again"
004012E0	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner = 7FFD6000
004012E3	E8 B4000000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004012E8	> 68 00020000	PUSH 200	Length = 200 (512.)
004012ED	68 49654000	PUSH crackme.00406549	Destination = crackme.00406549
004012F2	E8 ED000000	CALL <JMP.&kernel32.RtlZeroMemory>	RtlZeroMemory
004012F7	68 00020000	PUSH 200	Length = 200 (512.)

Here, we can see where the target calls the badboy. First, at address 4012D4, we see the value of 10h pushed on to the stack. Looking to the right where Olly has given us some labels, we see that this is the type of buttons, icons and style that this message box will be, namely a modal dialog with an OK button and an asterisk as the icon.

Next, we push the address for the memory that contains the caption (title) of the box. Here, we can push anything we want, so we'll just push the same thing they did. If we wanted to be fancy, we could create our own string to display for the caption, but here we'll just stick with what we've got.

Next we push the address of the memory that contains our actual text to be displayed in the box. This is where we'll push the address of the correct serial. Looking back where we dumped earlier, we see that this address is 406749.

Lastly, we push the handle for the owner's window. In this case, we'll just push the handle that was originally pushed.

Finally, we will simply jump to the address of the call to MessageBoxA in the original target's code at address 4012E3:

We will jump back to here:

```

00401200  E8 85844000  PUSH crackme.00406306
0040120B  FF 7624000  PUSH crackme.004062E7
00401218  FF 75 08    PUSH DWORD PTR SS:[EBP+8]
0040121E  E8 84000000  CALL <JMP.&user32.MessageBoxA>
00401225  2000       PUSH 2000
0040122D  E8 49654000  PUSH crackme.00406549
00401234  E8 00000000  PUSH <JMP.&kernel32.RtlZeroMemory>
0040123F  E8 00000000  PUSH 0

```

The screenshot shows the 'Assemble at 004041B0' dialog box. The address field contains 'push 10'. The 'Fill with NOP's' checkbox is checked. The 'Assemble' button is highlighted.

Assemble at 004041B2

push 10

☒ Fill with NOP's

Assemble Cancel

The screenshot shows a debugger window with a list of assembly instructions. The instruction at address 004041B7 is highlighted in red. A dialog box titled "Assemble at 004041BF" is open, showing the instruction "push dword ptr ss:[ebp+8]" in the input field. The dialog box has a "Fill with NOP's" checkbox checked, and "Assemble" and "Cancel" buttons.

Address	Disassembly	Comment
004041A0	DB 00	
004041A1	DB 00	
004041A2	DB 00	
004041A3	DB 00	
004041A4	DB 00	
004041A5	DB 00	
004041A6	DB 00	
004041A7	DB 00	
004041A8	DB 00	
004041A9	DB 00	
004041AA	DB 00	
004041AB	DB 00	
004041AC	DB 00	
004041AD	DB 00	
004041AE	DB 00	
004041AF	DB 00	
004041B0	PUSH 10	
004041B1	PUSH crackme.00406306	
004041B2	PUSH crackme.00406749	
004041B3	PUSH DWORD PTR SS:[EBP+8]	
004041B4	DB 00	
004041B5	DB 00	
004041B6	DB 00	
004041B7	PUSH 10	
004041B8	PUSH crackme.00406306	
004041B9	PUSH crackme.00406749	
004041BA	PUSH DWORD PTR SS:[EBP+8]	
004041BB	DB 00	
004041BC	DB 00	
004041BD	DB 00	
004041BE	DB 00	
004041BF	DB 00	
004041C0	DB 00	
004041C1	DB 00	
004041C2	DB 00	
004041C3	DB 00	
004041C4	DB 00	
004041C5	DB 00	
004041C6	DB 00	
004041C7	DB 00	
004041C8	DB 00	
004041C9	DB 00	
004041CA	DB 00	
004041CB	DB 00	
004041CC	DB 00	
004041CD	DB 00	
004041CE	DB 00	
004041CF	DB 00	
004041D0	DB 00	

ASCII "Bad boy..."

Assemble at 004041BF

push dword ptr ss:[ebp+8]

☒ Fill with NOP's

Assemble Cancel

004041AE	00	DB 00	ASCII "Bad boy..."
004041AF	00	DB 00	
004041B0	6A 10	PUSH 10	
004041B2	68 00634000	PUSH crackme.00406306	
004041B7	68 49674000	PUSH crackme.00406749	
004041BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004041BF	^ E9 96CFFFFF	JMP DWORD PTR SS:[EBP+8]	
004041C4	00	DB 00	
004041C5	00	DB 00	
004041C6	00	DB 00	
004041C7	00	DB 00	
004041C8	00	DB 00	

Here, we added the jump to 40115A, back to the original code's call to MessageBoxA.

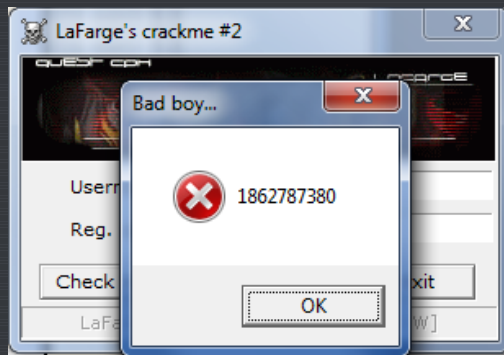
Now that we have our cave set up, we need to jump to it. Press cancel to close the assemble window, go back to address 401288, select the line and hit the space bar. Enter "jmp 4041B0". This jumps to our cave:

0040127E	88B 49674000	MOV BYTE PTR DS:[EBX+406749],CL	
00401284	43	INC EBX	
00401285	48	DEC EAX	
00401286	75 F0	JNZ SHORT crackme.00401278	
00401288	E9 232F0000	JMP crackme.004041B0	kernel32.BaseThreadInitThunk
0040128D	68 49654000	PUSH crackme.00406549	String1 = crackme.00406549
00401292	5F010000	CALL <JMP.&kernel32.lstrcpYA>	lstrcpYA
00401297	68 00020000	PUSH 200	Count = 200 (512.)

Let's try it out. Set a breakpoint at address 401288 (the jump to our cave) and run the app. Enter a username and any serial and click "Check it!". I entered "R4ndom" (of course). Olly should now break at the line that jumps to the code cave. Step into this jump and we will land at our cave:

004041AF	00	DB 00	
004041B0	6A 10	PUSH 10	
004041B2	68 06634000	PUSH crackme.00406306	ASCII "Bad boy..."
004041B7	68 49674000	PUSH crackme.00406749	ASCII "1862787380"
004041BC	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004041BF	E9 96CFFFFF	JMP crackme.0040115A	
004041C4	00	DB 00	
004041C5	00	DB 00	

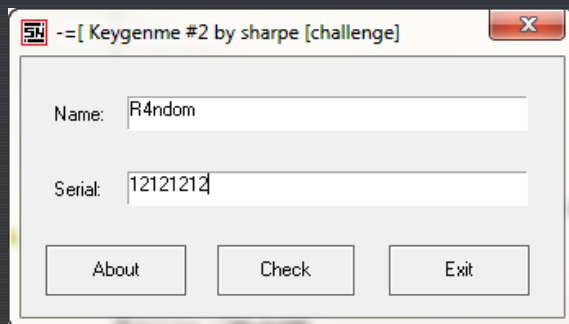
You will notice that Olly is displaying the correct serial at address 4041B7, so we know we're on the right track. Single stepping over the code, you will see the proper arguments pushed on to the stack, and finally, our jump back to the target's code. Once you step over the call to MessageBoxA, you will see our correct serial in the message box:



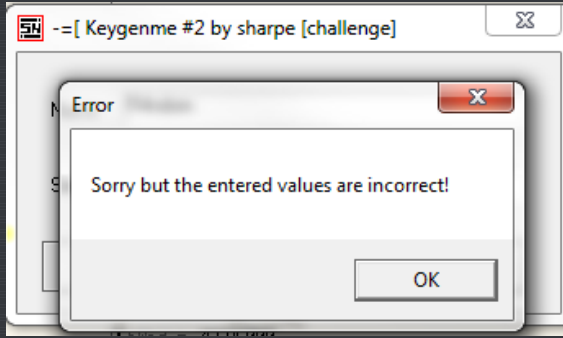
We have now created our own keygenner! You should save the entire binary back to the executable to save it so you have it stored on disk. Of course, if we were going to send this out into the wild, we would want to clean it up a bit, adding a real caption and a description of what this box is telling us, but for now, it's a beautiful thing.

Our Second Code Cave

Now let's try a little tougher example. Load up Keygenme.exe into Olly and run it:



and clicking the "Check" button:



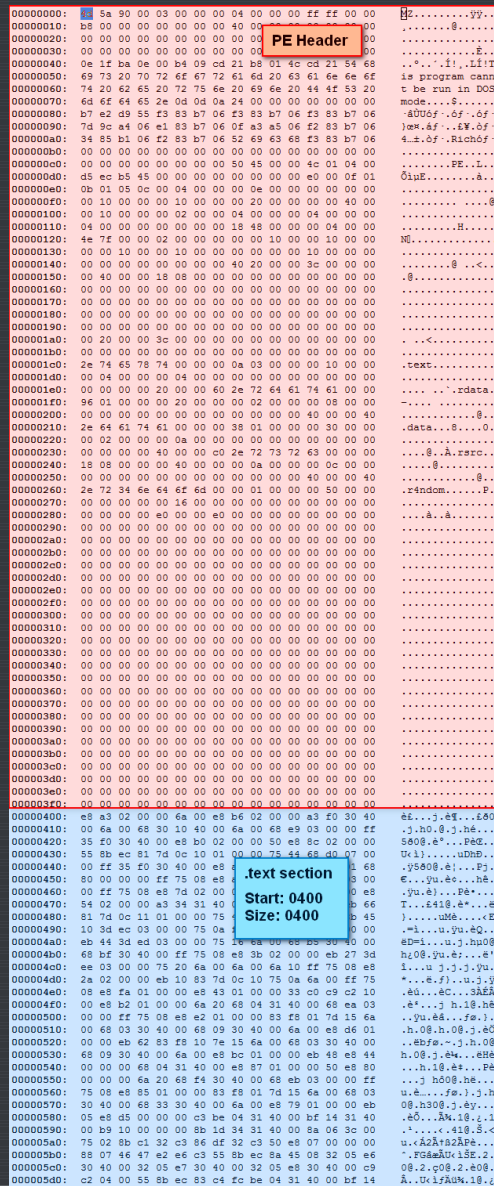
Well, that seems pretty straight forward, however this time we are going to assume that there is not enough space in the binary to insert a code cave, so we are going to make our own section.

*** In a future tutorial, we will be making this binary into a full-fledged keygen, so stay tuned. ***

Looking at The PE File Sections

This keygenme does not technically require a new section, but I wanted to go over the process, not only to understand how to manually add a section, but also to provide insight into how a PE file is structured. This will help further down the road when working with packers, malware, dll injection and so forth.

Here is an image of the complete keygen file, dumped from a hex editor. You can click on the image if you want to see it bigger:



```
data
start: 0800
size: 0200
```

```
.data
Start: 0A00
Size: 0200
```

```

.rsrc
Start: 0C00
Size: 0A00

```

00000f90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000010f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001120: 00 00 00 00 00 00 00 00 00 00 01 01 01 01 01 01
00001130: 01 01 01 01 01 01 01 00 00 01 02 01 02 02 02 02
00001140: 02 02 02 02 02 02 01 00 00 01 02 01 02 02 02 02
00001150: 02 02 02 02 02 02 01 00 00 01 01 01 01 01 01 01
00001160: 01 01 01 01 01 01 01 00 00 01 02 02 02 02 02 02
00001170: 01 02 01 01 02 02 01 00 00 01 01 01 01 02 02
00001180: 01 02 01 01 02 02 01 00 00 01 01 01 01 02 02
00001190: 01 02 02 02 02 02 01 00 00 01 02 02 02 02 02 02
000011a0: 01 02 02 01 01 02 01 00 00 01 02 02 01 01 01 01
000011b0: 01 02 01 01 02 01 00 00 01 02 02 02 02 02 02
000011c0: 01 02 02 01 01 02 01 00 00 01 01 01 01 01 01
000011d0: 01 01 01 01 01 01 01 00 00 01 02 02 02 02 02 02
000011e0: 02 02 02 01 02 01 00 00 01 02 02 02 02 02 02
000011f0: 02 02 02 02 01 02 01 00 00 01 01 01 01 01 01
00001200: 01 01 01 01 01 01 01 00 00 00 00 00 00 00 00 00
00001210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001250: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00
00001260: 00 00 01 00 08 00 68 05 00 00 01 00 00 00 00
00001270: 01 00 ff ff 00 00 00 00 00 00 00 00 40 08 cc 10
00001280: 07 00 00 00 00 00 dd 00 62 00 00 00 00 00 24 00
00001290: 3d 30 30 20 00 4b 00 65 00 65 00 79 00 67 00 65 00
000012a0: 6e 00 6d 00 65 00 20 00 23 00 32 00 20 00 62 00
000012b0: 79 00 20 00 73 00 68 00 61 00 72 00 70 00 65 00
000012c0: 20 00 5b 00 63 00 68 00 61 00 6c 00 6c 00 65 00
000012d0: 6e 00 67 00 65 00 5d 00 00 00 08 00 00 00 00 01
000012e0: 4d 00 53 00 20 00 53 00 61 00 6e 00 73 00 20 00
000012f0: 53 00 65 00 72 00 69 00 66 00 00 00 00 00 00 00
00001300: 00 00 02 00 00 00 00 50 2b 00 0f 00 a3 00 0d 00
00001310: ea 03 00 00 ff ff 81 00 00 00 00 00 00 00 00 00
00001320: 00 00 00 00 00 00 00 01 50 00 47 00 38 00 13 00
00001330: ec 03 00 ff ff 80 00 43 00 68 00 65 00 63 00 00
00001340: 6b 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001350: 00 80 01 50 0a 00 47 00 39 00 13 00 ed 03 00 00
00001360: ff ff 80 00 41 00 62 00 6f 00 75 00 74 00 00 00
00001370: 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 50
00001380: 0d 00 0f 00 15 00 0d 00 ef 03 00 00 ff ff 82 00
00001390: 4e 00 61 00 6d 00 65 00 3a 00 00 00 00 00 00 00
000013a0: 00 00 00 00 00 00 00 00 00 00 02 00 50 0d 00 2b 00
000013b0: 15 00 0d 00 20 03 00 00 ff ff 82 00 53 00 65 00
000013c0: 72 00 69 00 61 00 6e 00 3a 00 00 00 00 00 00 00
000013d0: 00 00 00 00 00 00 02 00 00 00 00 50 2b 00 2b 00
000013e0: a3 00 0d 00 eb 03 00 00 ff ff 81 00 00 00 00 00 00
000013f0: 00 00 00 00 00 00 00 00 00 80 01 50 96 00 47 00
00001400: 39 10 00 00 00 00 ff ff 80 00 45 00 78 00 00 00
00001410: 69 00 74 00 00 00 00 00 00 00 00 00 00 00 00
00001420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001490: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000014f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001510: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001520: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001530: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001540: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001590: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000015a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000015b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000015c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000015d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000015e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

We can match these sections up with the graph shown in CFF Explorer, under “Section Headers”:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000030A	00001000	00000400	00000400	00000000	00000000	0000	0000	60000020
.rdata	00000196	00002000	00000200	00000800	00000000	00000000	0000	0000	40000040
.data	00000138	00003000	00000200	00000A00	00000000	00000000	0000	0000	C0000040
.rsrc	00000818	00004000	00000A00	00000C00	00000000	00000000	0000	0000	40000040

Notice this binary has four sections: .text where our code goes, .rdata for our read-only data (strings and whatnot), .data for our read-write data (global variables) and .rsrc where our resources are stored (buttons, bitmaps, dialogs...). There is also one fifth section, the PE header, shown at the top of the dump. This is not shown as a section, but in the binary, it could be thought of as one.

The start addresses and sizes are shown in the Raw Size and Raw Address columns. This signifies where the data is on disk in its raw form. We can see in the CFF graph that the .text section starts at 0400 in the binary and has a length of 0400. Looking up at the dump, starting at address 0400, we see the .text section begins and ends at address 07FF. This makes the size 0400 bytes.

Each section has a starting address and a size in the 'raw' columns, telling us where they reside in the

binary. When the Windows loader loads this binary into memory, it copies these sections from disk and places them into memory. The 'virtual' columns represents where this data will be copied into memory and how large the space will occupy. We can see in the .text section the beginning address will be 01000 and the size of the copied data will be 030A. This means that when the loader loads this binary from disk, it will load the data beginning at 0400 in the binary into memory beginning at memory address 01000. It will copy 030A bytes from disk, which is smaller than the region allocated in memory, so there will be 0CF6 bytes at the end of this section with zeroes in it (01000 is the size of this memory space, 30A the size of the data. $01000 - 030A = 0CF6$).

You may say, "Wait a minute, why is the binary always loaded into address 401000 in Olly, and not at address 01000?" The reason has to do with a field in the PE header called ImageBase. Clicking on the "Optional Header" in CFF, we see that the ImageBase is set to 040000:

keygenme.exe				
Member	Offset	Size	Value	Meaning
Magic	000000E0	Word	010B	PE
MajorLinkerVersion	000000E2	Byte	05	
MinorLinkerVersion	000000E3	Byte	0C	
SizeOfCode	000000E4	Dword	00000400	
SizeOfInitializedData	000000E8	Dword	00000E00	
SizeOfUninitializedData	000000EC	Dword	00000000	
AddressOfEntryPoint	000000F0	Dword	00001000	.text
BaseOfCode	000000F4	Dword	00001000	
BaseOfData	000000F8	Dword	00002000	
ImageBase	000000FC	Dword	00400000	
SectionAlignment	00000100	Dword	00001000	

this tells the loader to load in all sections, but start at address 040000. The Windows loader does not have to load the binary into this address- it is simply where the application *would like* to be placed. Most of the time, the loader will comply.

*** If you have the randomize base address setting enabled in Visual Studio, the base address will be random, so it will probably not be loaded at address 401000. You can always tell if this has been set in a binary when you load it in Olly and the beginning of the binary is at some weird starting address, such as 0x1143679. ***

The next section, .data, which begins on disk at offset 0800 (and a size of 0200) will be copied into memory beginning at memory address 02000, and 0196 bytes will be copied. Since the next section begins at 03000, this section also has a size in memory of 01000 bytes, so 0804 bytes will be left at the end.

The section at the end of the .text section is where we would normally put our code cave, and in this case it would be fine. The problem arises when the code on disk is almost the same size as the memory space allotted for this code. If the Loader reserves 01000 bytes for the code, and the code takes up 0988 bytes, this would only leave 012 bytes for our cave. Because this would not be enough room, we must add a section to put our cave into.

Manually Adding a New Section

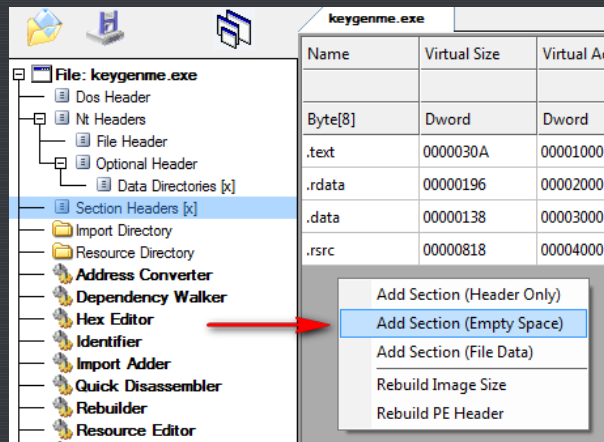
We are going to use CFF Explorer to add our section. Go ahead and load the keygenme into CFF if you haven't already and click on the "File Header" tab:

File: keygenme.exe					
Machine	000000CC	Word	014C	Intel 386	
NumberOfSections	000000CE	Word	0004		
TimeDateStamp	000000D0	Dword	45B5ECD5		
PointerToSymbolTable	000000D4	Dword	00000000		
NumberOfSymbols	000000D8	Dword	00000000		
SizeOfOptionalHeader	000000DC	Word	00E0		
Characteristics	000000DE	Word	010F	Click here	

We can see that there are clearly four sections in this binary. Clicking on the "Section Headers" we see what we saw earlier, namely the information about each section:

File: keygenme.exe	Name	Virtual Size	Virtual Address	Raw Size
Dos Header				
Nt Headers				
File Header				
Optional Header				
Data Directories [x]				
Section Headers [x]	Byte[8]	Dword	Dword	Dword
Import Directory	.text	0000030A	00001000	00000400
Resource Directory	.rdata	00000196	00002000	00000800
Address Converter	.data	00000138	00003000	00000A00
	.rsrc	00000818	00004000	00000C00

Now right-click in this window and choose "Add Section (Empty Space)" :



In the space dialog, enter 100h, as we wish our new section to have 100 (256) bytes:

Size of section

100h

Put a space and a d(ec) or an h(ex) after the number (if you have to enter one) to specify the type (e.g. "123 d"). By default numbers are considered hex.

OK Cancel

Clicking OK we will see our new section show up:

Byte[8]	Dword	Dword	Dword	Dword
.text	0000030A	00001000	00000400	00000400
.rdata	00000196	00002000	00000200	00000800
.data	00000138	00003000	00000200	00000A00
.rsrc	00000818	00004000	00000A00	00000C00
	00000100	00005000	00000200	00001600

Let's rename our section. Double-click in the name field and enter ".r4ndom" and click return:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000030A	00001000	00000400	00000400	00000000	00000000	0000	0000	60000020
.rdata	00000196	00002000	00000200	00000800	00000000	00000000	0000	0000	40000040
.data	00000138	00003000	00000200	00000A00	00000000	00000000	0000	0000	C0000040
.rsrc	00000818	00004000	00000A00	00000C00	00000000	00000000	0000	0000	40000040
.r4ndom	00000100	00005000	00000200	00001600	00000000	00000000	0000	0000	C0000000

Also notice that the initial size values have been provided; The virtual size is 0100, the virtual address is

05000, the raw size is 0200 and the raw address is 01600. Let's take these one at a time:

VirtualSize: This is the amount we placed in the size dialog.

VirtualAddress: Notice that the default size for a section is 01000 bytes in memory. This value is set in the "SectionAlignment" field in the PE header.

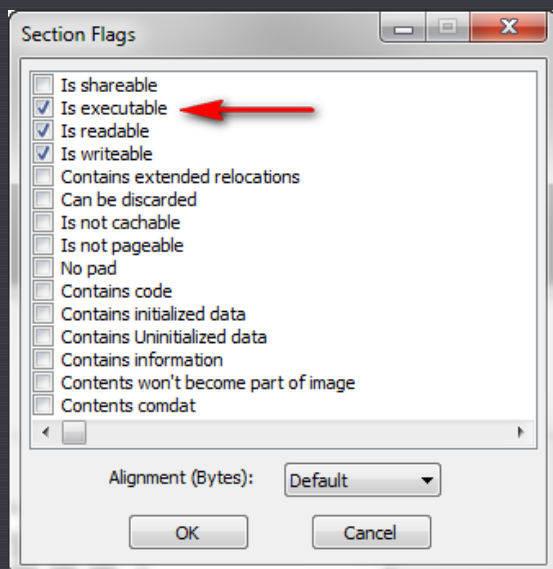
RawSize: Notice that the size on disk is 0200, not 0100 which we would expect. This is because there is a minimum amount of space that a section can take on disk. This value is set in the "FileAlignment" field in the PE header.

RawAddress: If you look back at our hex dump of the binary, you will see that it originally ended at 015FF. Logically, our new section would start right after this, at address 01600.

keygenme.exe			
Member	Offset	Size	Value
Magic	000000E0	Word	010B
MajorLinkerVersion	000000E2	Byte	05
MinorLinkerVersion	000000E3	Byte	0C
SizeOfCode	000000E4	Dword	00000400
SizeOfInitializedData	000000E8	Dword	00000E00
SizeOfUninitializedData	000000EC	Dword	00000000
AddressOfEntryPoint	000000F0	Dword	00001000
BaseOfCode	000000F4	Dword	00001000
BaseOfData	000000F8	Dword	00002000
ImageBase	000000FC	Dword	00400000
SectionAlignment	00000100	Dword	00001000
FileAlignment	00000104	Dword	00000200
MajorOperatingSystemVers...	00000108	Word	0004

Here we can see the memory and disk alignments.

One last setting we need is to make our new section executable, meaning we need to allow code to run in it, as the default for a new section is for it just to be data. Right-click the new section and select "Change Section Flags." We want to set the "Is Executable" flag:



Now our code cave will be able to run in memory.

Before we can add our section, we must update the size of the binary (as we've added 0100 bytes) as well as the header. Right-click the line with our new section on it and select "Rebuild Image Size". This will add 0200 bytes on to the total size of the binary. Next, right-click and choose "Rebuild PE Header." This updates our number of sections field, as well as some other fields necessary to load the new section:

Member	Offset	Size	Value	Meaning
Machine	000000CC	Word	014C	Intel 386
NumberOfSections	000000CE	Word	0005	
TimeStamp	000000D0	Dword	45B5ECD5	
PointerToSymbolT...	000000D4	Dword	00	
NumberOfSymbols	000000D8	Dword	00	
SizeOfOptionalHea...	000000DC	Word	00E0	
Characteristics	000000DE	Word	010F	Click here

Number of sections
is now "5"

Finally, we must save our changes. Click "File" -> "Save As" and choose "keygenme2.exe" for the name. Now load the new file into Olly and bring up the Memory Window:

Memory map					
Address	Size	Owner	Section	Contains	Type
00030000	00001000				Priv 00021004
00040000	00001000	apisetsc			Imag 01001002
00089000	00007000				Priv 00021104
0018C000	00001000				Priv 00021104
0018D000	00003000				Priv 00021104
00190000	00004000				Map 00041002
001A0000	00001000				Priv 00021004
001B0000	00003000				Priv 00021004
00230000	0004B000				Priv 00021004
00330000	00067000				Map 00041002
00400000	00001000	keygenme		PE header	Imag 01001002
00401000	00001000	keygenme	.text	SFX, code	Imag 01001002
00402000	00001000	keygenme	.rdata	data, imports	Imag 01001002
00403000	00001000	keygenme	.data		Imag 01001002
00404000	00001000	keygenme	.rsrc	resources	Imag 01001002
00405000	00001000	keygenme	.r4ndom		Imag 01001002
00500000	00003000				Priv 00021004

We can see that Olly recognizes our new section, and we are now ready to investigate where we will call our code cave from...

Investigating the Target

OK. Let's take a look at our target and figure out where to call our cave. Return to the entry point (ctrl-G and enter 401000) and do a search for intermodular calls:

Found intermodular calls		
Address	Disassembly	Destination
00401000	JMP 71B00000	(Initial CPU selection)
00401025	CALL <JMP.&user32.ShowDialogParamA>	user32.ShowDialogParamA
004010E1	CALL <JMP.&user32.EndDialog>	user32.EndDialog
004010E8	CALL <JMP.&kernel32.ExitProcess>	kernel32.ExitProcess
004012B6	CALL <JMP.&kernel32.ExitProcess>	kernel32.ExitProcess
00401064	CALL <JMP.&user32.GetDlgItem>	user32.GetDlgItem
00401105	CALL <JMP.&user32.GetDlgItemTextA>	user32.GetDlgItemTextA
00401162	CALL <JMP.&user32.GetDlgItemTextA>	user32.GetDlgItemTextA
00401007	CALL <JMP.&kernel32.GetModuleHandleA>	kernel32.GetModuleHandleA
0040106F	CALL <JMP.&kernel32.GetVersion>	kernel32.GetVersion
004012AA	CALL <JMP.&kernel32.IsDebuggerPresent>	kernel32.IsDebuggerPresent
00401047	CALL <JMP.&user32.LoadIconA>	user32.LoadIconA
00401148	CALL <JMP.&kernel32.lstrlenA>	kernel32.lstrlenA
004010B8	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
0040111D	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
00401137	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
0040117A	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
004012B0	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
004012A2	CALL <JMP.&user32.MessageBoxA>	user32.MessageBoxA
00401057	CALL <JMP.&user32.SendMessageA>	user32.SendMessageA
004010CF	CALL <JMP.&user32.SendMessageA>	user32.SendMessageA
0040106A	CALL <JMP.&user32.SetFocus>	user32.SetFocus

The GetDlgItemTextA looks as good as any. Click on the first and we see we're in the general area:

004010E1	CALL E2010000	CALL Keygenme.004012A8	Count = 20 (32.)
004010F1	6A 20	PUSH 0	Buffer = Keygenme.00403104
004010F6	68 04314000	PUSH Keygenme.00403104	ControlID = 3EA (1002.)
004010F8	68 E8030000	PUSH 3EA	hWnd = 7EFDE000
004010F9	FF75 08	PUSH [ARG_1]	GetDlgItemTextA
00401105	E8 E2010000	CALL <JMP.>user32.GetDlgItemTextA	GetDlgItemTextA
0040110A	83F8 01	CMF EAX,1	
0040110D	7D 15	JGE SHORT Keygenme.00401124	
0040110F	6A 00	PUSH 0	Style = MB_OK!MB_APPLMODAL
00401111	68 03304000	PUSH Keygenme.00403003	Title = "Error"
00401116	68 09304000	PUSH Keygenme.00403009	Text = "Your name must be between 1 and 16 bytes!"
00401118	6A 00	PUSH 0	hOwner = NULL
0040111D	E8 D6010000	CALL <JMP.>user32.MessageBoxA	MessageBoxA
00401122	EB 62	JMP SHORT Keygenme.00401186	
00401124	33F8 10	CMF EAX,10	
00401127	7E 15	JLE SHORT Keygenme.0040113E	
00401129	6A 00	PUSH 0	Style = MB_OK!MB_APPLMODAL
0040112B	68 03304000	PUSH Keygenme.00403003	Title = "Error"
00401130	68 09304000	PUSH Keygenme.00403009	Text = "Your name must be between 1 and 16 bytes!"
00401135	6A 00	PUSH 0	hOwner = NULL
00401137	E8 BC010000	CALL <JMP.>user32.MessageBoxA	MessageBoxA
0040113C	EB 48	JMP SHORT Keygenme.00401186	
0040113E	E8 44000000	CALL Keygenme.00401187	
00401143	68 04314000	PUSH Keygenme.00403104	String = ""
00401148	E8 87010000	CALL <JMP.>kernel32.lstrlenA	lstrlenA
0040114D	50	PUSH EAX	Arg1 = 76323398
0040114E	E8 80000000	CALL Keygenme.004011D3	Keygenme.004011D3
00401153	6A 20	PUSH 20	Count = 20 (32.)
00401155	68 F4304000	PUSH Keygenme.004030F4	Buffer = Keygenme.004030F4
0040115A	68 E8030000	PUSH 3EB	ControlID = 3EB (1003.)
0040115F	FF75 08	PUSH [ARG_1]	Count = 20 (32.)
00401162	E8 85010000	CALL <JMP.>user32.GetDlgItemTextA	Buffer = Keygenme.004030F4
00401167	83F8 01	CMF EAX,1	ControlID = 3EB (1003.)
0040116A	7D 15	JGE SHORT Keygenme.00401181	hWnd = 7EFDE000
0040116C	6A 00	PUSH 0	GetDlgItemTextA
0040116E	68 03304000	PUSH Keygenme.00403003	Style = MB_OK!MB_APPLMODAL
00401173	68 33304000	PUSH Keygenme.00403033	Title = "Error"
00401178	6A 00	PUSH 0	Text = "Your serial must be at least one byte!"
0040117A	E8 79010000	CALL <JMP.>user32.MessageBoxA	hOwner = NULL
0040117F	EB 05	JMP SHORT Keygenme.00401186	MessageBoxA
00401181	E8 D5000000	CALL Keygenme.0040125B	
00401186	C3	RETN	
00401187	BE 04314000	MOV ESI,Keygenme.00403104	

We can see that at first the target gets our entered user name and stores it at address 403105 (this value is pushed at address 4010F8), then checks it for length. Obviously, it needs to be between 1 and 16 characters. After that, the target checks to make sure we entered at least one character in the serial field:

0040114E	E8 80000000	CALL Keygenme.004011D3	Keygenme.004011D3
00401153	6A 20	PUSH 20	Count = 20 (32.)
00401155	68 F4304000	PUSH Keygenme.004030F4	Buffer = Keygenme.004030F4
0040115A	68 E8030000	PUSH 3EB	ControlID = 3EB (1003.)
0040115F	FF75 08	PUSH [ARG_1]	Count = 20 (32.)
00401162	E8 85010000	CALL <JMP.>user32.GetDlgItemTextA	Buffer = Keygenme.004030F4
00401167	83F8 01	CMF EAX,1	ControlID = 3EB (1003.)
0040116A	7D 15	JGE SHORT Keygenme.00401181	hWnd = 7EFDE000
0040116C	6A 00	PUSH 0	GetDlgItemTextA
0040116E	68 03304000	PUSH Keygenme.00403003	Style = MB_OK!MB_APPLMODAL
00401173	68 33304000	PUSH Keygenme.00403033	Title = "Error"
00401178	6A 00	PUSH 0	Text = "Your serial must be at least one byte!"
0040117A	E8 79010000	CALL <JMP.>user32.MessageBoxA	hOwner = NULL
0040117F	EB 05	JMP SHORT Keygenme.00401186	MessageBoxA
00401181	E8 D5000000	CALL Keygenme.0040125B	
00401186	C3	RETN	
00401187	BE 04314000	MOV ESI,Keygenme.00403104	

But you may have noticed there was a quite little call between these two code groups at address 40113E. This is the call to our encryption routine:

00401130	68 09304000	PUSH Keygenme.00403009	Text = "Your name must be between 1 and 16 bytes!"
00401135	6A 00	PUSH 0	hOwner = NULL
00401137	E8 BC010000	CALL <JMP.>user32.MessageBoxA	MessageBoxA
0040113C	EB 48	JMP SHORT Keygenme.00401186	
0040113E	E8 44000000	CALL Keygenme.00401187	String = "R4ndom"
00401143	68 04314000	PUSH Keygenme.00403104	lstrlenA
00401148	E8 87010000	CALL <JMP.>kernel32.lstrlenA	Arg1 = 00000006
0040114D	50	PUSH EAX	Keygenme.004011D3
0040114E	E8 80000000	CALL Keygenme.004011D3	Count = 20 (32.)
00401153	6A 20	PUSH 20	Buffer = Keygenme.004030F4
00401155	68 F4304000	PUSH Keygenme.004030F4	ControlID = 3EB (1003.)
0040115A	68 E8030000	PUSH 3EB	hWnd = 000C09AE ('-=[Keygenme #2 by sharpe [ch...')
0040115F	FF75 08	PUSH [ARG_1]	GetDlgItemTextA
00401162	E8 85010000	CALL <JMP.>user32.GetDlgItemTextA	GetDlgItemTextA
00401167	83F8 01	CMF EAX,1	
0040116A	7D 15	JGE SHORT Keygenme.00401181	Style = MB_OK!MB_APPLMODAL
0040116C	6A 00	PUSH 0	Title = "Error"
0040116E	68 03304000	PUSH Keygenme.00403003	Text = "Your serial must be at least one byte!"
00401173	68 33304000	PUSH Keygenme.00403033	hOwner = NULL
00401178	6A 00	PUSH 0	MessageBoxA
0040117A	E8 79010000	CALL <JMP.>user32.MessageBoxA	
0040117F	EB 05	JMP SHORT Keygenme.00401186	ASCII "R4ndom"
00401181	E8 D5000000	CALL Keygenme.0040125B	
00401186	C3	RETN	
00401187	BE 04314000	MOV ESI,Keygenme.00403104	
00401188	BF 14314000	MOV EDI,Keygenme.00403114	
00401191	B9 10000000	MOV ECX,10	
00401196	8B1D 34314000	MOV EBX,DWORD PTR DS:[403134]	
0040119C	8A06	MOV AL,BYTE PTR DS:[ESI]	
0040119E	3C 00	CMF AL,0	
004011A0	75 02	JNZ SHORT Keygenme.004011A4	
004011A2	8BC1	MOV EAX,ECX	
004011A4	32C3	XOR AL,BL	
004011A6	86DF	XCHG BH,BL	
004011A8	32C3	XOR AL,BL	
004011AA	50	PUSH EAX	
004011AB	E8 07000000	CALL Keygenme.004011B7	
004011B0	8B07	MOV BYTE PTR DS:[EDI],AL	
004011B2	46	INC ESI	
004011B3	47	INC EDI	
004011B4	E2 E6	LOOPD SHORT Keygenme.0040119C	
004011B6	C3	RETN	
004011B7	55	PUSH EBP	
004011B8	8BEC	MOV EBP,ESP	
004011BA	8A45 08	MOV AL,BYTE PTR SS:[EBP+8]	
004011BD	3205 E6304000	XOR AL,BYTE PTR DS:[4030E6]	
004011C3	3205 E7304000	XOR AL,BYTE PTR DS:[4030E7]	
004011C9	3205 E8304000	XOR AL,BYTE PTR DS:[4030E8]	
004011CF	C9	LEAVE	
004011D0	C2 0400	RETN 4	
004011D3	55	PUSH EBP	
004011D4	8BEC	MOV EBP,ESP	
004011D6	83C4 FC	ADD ESP,-4	
004011D9	BE 04314000	MOV ESI,Keygenme.00403104	
004011DE	BF 14314000	MOV EDI,Keygenme.00403114	
004011E3	B9 10000000	MOV ECX,10	
004011E8	C645 FF 00	MOV BYTE PTR SS:[EBP-3],0	

Go ahead and place a breakpoint at this address (40113E) and run the target. Enter a Name (R4ndom) and a serial (doesn't matter) and click "Check" and Olly breaks at our breakpoint. Now, before we

continue, let's load the memory where our name was stored in the dump. Go to address 4010F8, right-click and select "Follow in dump"->"Immediate constant". Now we see our username in the dump:

Address	Hex dump	ASCII
00403104	52 34 6E 64 6F 6D 00 00 00 00 00 00 00 00 00 00	R4ndom.....
00403114	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403124	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403134	06 01 B1 1D 00 00 00 00 00 00 00 00 00 00 00 00	*C##.....
00403144	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403184	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Now let's single step over the call at address 40113E and we will see some strange characters appear in our dump:

Address	Hex dump	ASCII
00403104	52 34 6E 64 6F 6D 00 00 00 00 00 00 00 00 00 00	R4ndom.....
00403114	06 B0 EA E0 EB E9 8E 8D 8C 83 82 81 80 87 86 85	~@~\$0A1a~u~C~a
00403124	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403134	06 01 B1 1D 00 00 00 00 00 00 00 00 00 00 00 00	*C##.....
00403144	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403174	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

We can safely assume that this is the first part of the encryption routine. Single-stepping down a couple lines to address 40114E, we see that there is another call to this encryption section. Stepping over this line, we see the weird characters in the dump turn into the correct serial:

Address	Hex dump	ASCII
00403104	52 34 6E 64 6F 6D 00 00 00 00 00 00 00 00 00 00	R4ndom.....
00403114	45 30 43 32 36 32 36 37 38 39 41 45 39 42 41 44	E0C2626789AE9BAD
00403124	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403134	06 01 B1 1D 00 00 00 00 00 00 00 00 00 00 00 00	*C##.....
00403144	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403154	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403164	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

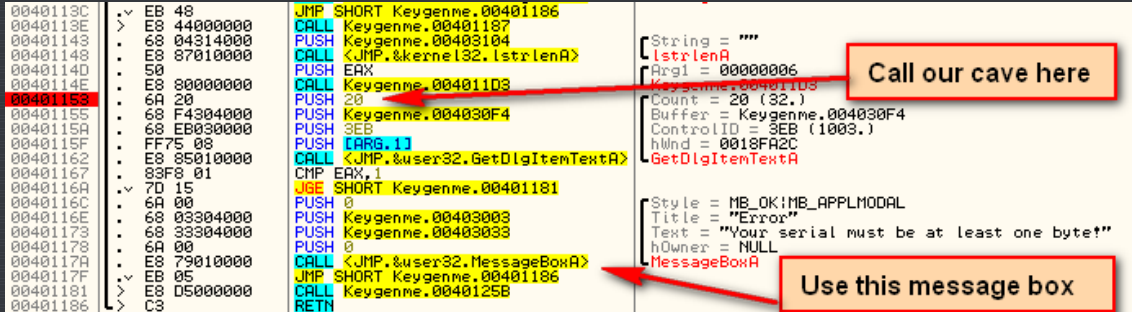
*** You may ask, "Is it always this easy"? No. We got lucky on this one. Though it would not have been very difficult to step through the encryption code to find out where this serial was being stored, which is exactly what I did when first going through this crackme. ***

Now we know (or at least assume) that the correct serial is known by the time we get to address 40115A, and it is stored at location 403114. We can test our theory, though, just to be sure:

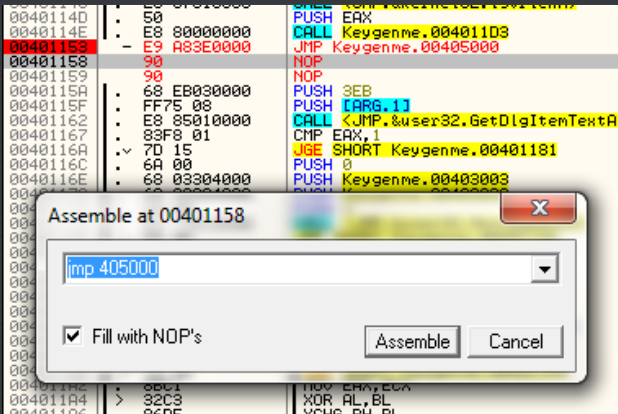
and we see we were correct:

Adding the Code Cave

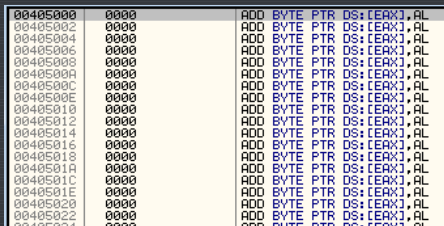
I have set a breakpoint at address 401153 and deleted my others. We know that at this point, memory location 403114 contains our correct serial. So we will place our call here. We are pretty fortunate here in that we also have a message box call right after our jump that we can use to show our serial:



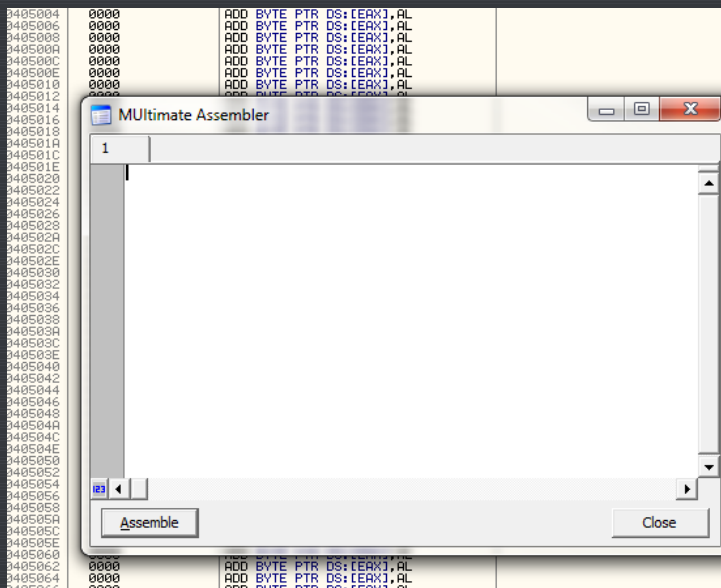
We know our code cave space starts at 405000, as this is the beginning of the section we created, so let's go ahead and add our jump:



Now let's create our cave. Make Olly display our new section by going to address 405000:



Now we are going to do the same thing we did in the first part of this tutorial, but we are going to use a new plugin called MULTimate Assembler. Make sure this has been copied into your Olly plugin's folder, and select "MULTimate Assembler"->"MULTimate Assembler" from the plugins drop-down menu:



*** One note: if you are using the OllyAdvanced plugin, you must turn off the "Kill %s%s bug" option or the

assembled code in multi-asm may not work properly. ***

This time, in order to show the ease to which you can assemble with this plugin, we will change the caption and text in our message box. Let's begin coding. First, MUlimate Assmsembler requires that we put in a starting address:

<405000>

I can put anything in here, so I put the beginning of our cave where the initial jump will jump to. Next I start coding the actual cave:

push 10

This pushes the message box style. Then we push our caption. I will define the caption string towards the end, but for now we can push the address:

push @caption

Next we push the text. This is the same deal as the caption:

push @text

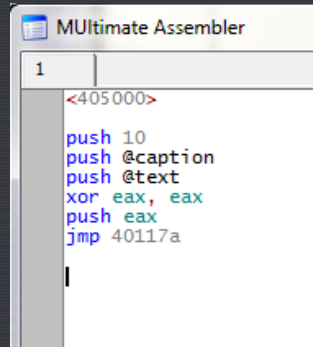
Finally, we push the handle. In this case, I'm just going to push NULL, as the message box call doesn't have to have a handle:

xor eax, eax
push eax

Now we will put our jump back to the original MessageBoxA call:

jmp 40117A

Your screen should look like this so far:



Next, let's add our caption:

@caption:
"Correct serial.\0"

and our message box text:

@text:
"The correct serial is: "

Notice I did not add a zero at the end of this string. this is because we are going to add the text for our serial after it. Message box will start at the beginning of this string and go until it reaches a zero. We are going to manually add the correct serial on to the end of this string and end that with a zero. This way, when message box is displaying the text, it will keep going into the string we add, and will not stop until after the correct serial is displayed as well. So next we must add space to copy in our correct serial:

@serial:
db 00

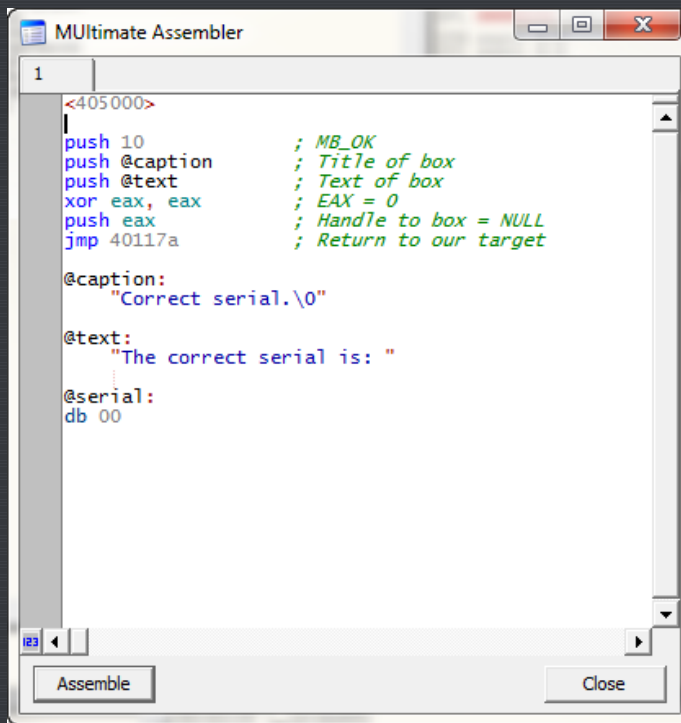
Here, I have added an empty byte that will be the first character of our serial. since I don't know the length of the serial, I simply put one byte here, and I will dynamically add the additional bytes of the correct serial,

followed by a zero.

*** The maximum number of bytes you can use in MUltimate Assembler after the 'db' command is 16. ***

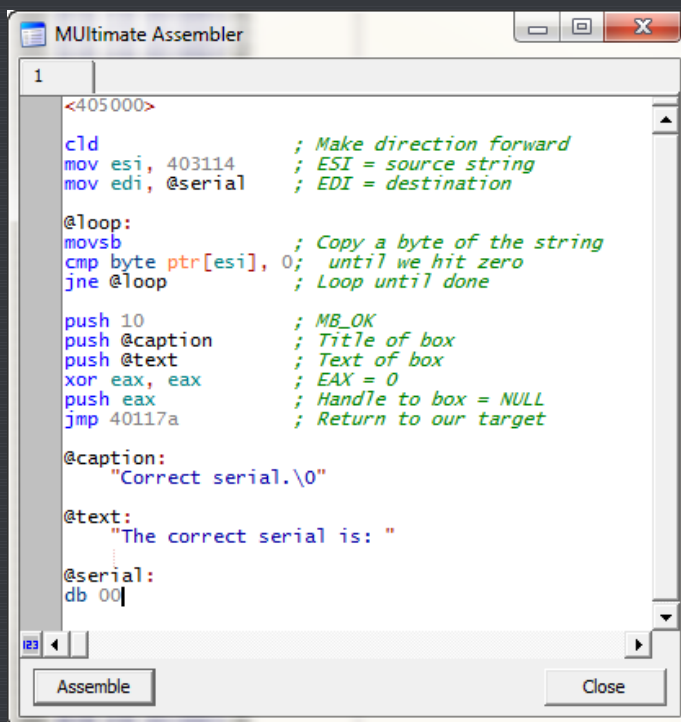
Because this string is at the end of our cave, I can add as many characters as I like to the end, as long as it ends with a zero (and does not go past the end of the section).

Alright, up to this point your cave should look like this (I also added some comments):



*** You may want to save your assembly at this point- right-click the main tab and select "Save to file". Now, if anything happens, you can just reload it. ***

Before we're done, we must copy the original serial into our string that will display the text in our message box. To do this, normally we would use `strcpy`, but since we can only use the functions that are available to the target (at least easily), and this binary does not use that function, we will do it by hand. Insert the following instructions at the beginning of our assembly window:



*** If you have any trouble, I have included the code for this code cave in the download for this tutorial.
Just open the Ultimate Assembler plugin and load the file "Keygenme.asm". ***

Go ahead and click "Assemble". If there are no errors, you will see the code in your binary assembled at 405000. If there is an error, the error type will be displayed and the line the error at will receive the cursor. If all went well, you should see something like this:

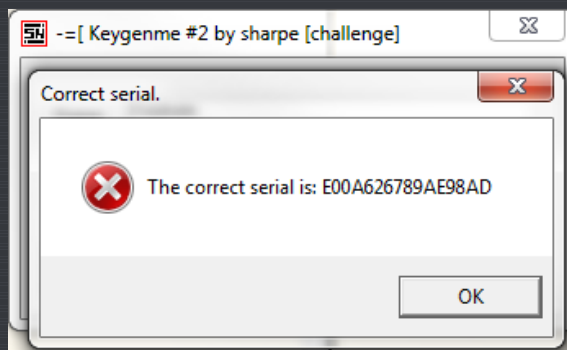
00405000	FC	CLD	Make direction forward
00405001	BE 14314000	MOV ESI,Keygenme.00403114	ESI = source string
00405006	BF 50504000	MOV EDI,<Keygenme.serial>	EDI = destination
0040500B	A4	MOVS BYTE PTR ES:[EDI],BYTE PTR DS:	Copy a byte of the string
0040500C	803E 00	CMP BYTE PTR DS:[ESI],0	until we hit zero
0040500F	^ 0F85 F6FFFFFF	JNZ <Keygenme.loop>	Loop until done
00405015	6A 10	PUSH 10	MB_OK
00405017	68 29504000	PUSH OFFSET <Keygenme.caption>	Title of box
0040501C	68 39504000	PUSH OFFSET <Keygenme.text>	Text of box
00405021	33C0	XOR EAX,EAX	EAX = 0
00405023	50	PUSH EAX	Handle to box = NULL
00405024	- E9 51C1FFFF	JMP Keygenme.0040117A	Return to our target
00405029	43	INC EBX	
0040502A	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
0040502B	72 72	JB SHORT Keygenme.0040509F	
0040502D	65:637420 73	ARPL WORD PTR GS:[EAX+73],SI	
00405032	65:72 69	JB SHORT Keygenme.0040509E	Superfluous prefix
00405035	61	POPAD	
00405036	6C	INS BYTE PTR ES:[EDI],DX	I/O command
00405037	2E:005468 65	ADD BYTE PTR CS:[EAX+EBP*2+65],DL	
0040503C	2063 6F	AND BYTE PTR DS:[EBX+6F],AH	
0040503F	72 72	JB SHORT Keygenme.004050B3	
00405041	65:637420 73	ARPL WORD PTR GS:[EAX+73],SI	
00405046	65:72 69	JB SHORT Keygenme.004050B2	Superfluous prefix
00405049	61	POPAD	
0040504A	6C	INS BYTE PTR ES:[EDI],DX	I/O command
0040504B	2069 73	AND BYTE PTR DS:[ECX+73],CH	
0040504E	3A20	CMP AH,BYTE PTR DS:[EAX]	
00405050	0000	ADD BYTE PTR DS:[EAX],AL	
00405052	0000	ADD BYTE PTR DS:[EAX],AL	
00405054	0000	ADD BYTE PTR DS:[EAX],AL	
00405056	0000	ADD BYTE PTR DS:[EAX],AL	
00405058	0000	ADD BYTE PTR DS:[EAX],AL	
0040505A	0000	ADD BYTE PTR DS:[EAX],AL	
0040505C	0000	ADD BYTE PTR DS:[EAX],AL	
0040505E	0000	ADD BYTE PTR DS:[EAX],AL	
00405060	0000	ADD BYTE PTR DS:[EAX],AL	
00405062	0000	ADD BYTE PTR DS:[EAX],AL	
00405064	0000	ADD BYTE PTR DS:[EAX],AL	
00405066	0000	ADD BYTE PTR DS:[EAX],AL	
00405068	0000	ADD BYTE PTR DS:[EAX],AL	

You will notice that Olly cannot figure out that the end of the cave is strings, so he is showing the data as code. Use the AnalyzeThis! plugin (Right-click->AnalyzeThis!) and Olly will get it straight:

00405000	FC	CLD	Make direction forward
00405001	BE 14314000	MOV ESI,Keygenme.00403114	ESI = source string
00405006	BF 50504000	MOV EDI,<Keygenme.serial>	EDI = destination
0040500B	> A4	MOVS BYTE PTR ES:[EDI],BYTE PTR DS:	Copy a byte of the string
0040500C	803E 00	CMP BYTE PTR DS:[ESI],0	until we hit zero
0040500F	^ 0F85 F6FFFFFF	JNZ <Keygenme.loop>	Loop until done
00405015	6A 10	PUSH 10	MB_OK
00405017	68 29504000	PUSH <Keygenme.caption>	Title of box
0040501C	68 39504000	PUSH <Keygenme.text>	Text of box
00405021	33C0	XOR EAX,EAX	EAX = 0
00405023	50	PUSH EAX	Handle to box = NULL
00405024	- E9 51C1FFFF	JMP Keygenme.0040117A	Return to our target
00405029	43 6F 72 72 65	ASCII "Correct serial.",0	
00405039	54 68 65 20 63	ASCII "The correct seri"	
00405049	61 6C 20 69 73	ASCII "al is: "	
00405050	00	DB 00	
00405051	00	DB 00	
00405052	00	DB 00	
00405053	00	DB 00	

Save our new patched file as "keygenme3.exe". Remember, when selecting "Copy to executable", choose "All modifications" so that the jump from address 401153 is included, as well as our code cave.

Now, go ahead and run the target. Enter your desired username and click "Check":



We now have our own keygenme! Well, sort of. In a future tutorial we will be creating an actual keygenme from this crackme, but for now, this works pretty well...

-Till next time

R4ndom