

Home

Tutorials

Tools

Contact

Forum

Challenges

R4ndom's Tutorial #21: Anti-Debugging Techniques

by R4ndom on Sep.14, 2012, under Beginner, Reverse Engineering, Tutorials

Nowadays, with the plethora of anti-anti-debugging plugins for Olly, you don't need to know nearly as much as you used to about anti-debugging techniques. But the problem is, without understanding how they work, when we are confronted with a new technique we find that we have no idea how to overcome it. Additionally, learning about anti-debugging techniques helps us understand low-level protections, and is a good introduction to packers.

Anti-debugging is a rather large field, and impossible to cover in one tutorial. I do hope to shed some light on the most used techniques, as well as direct you to getting additional information on some of the more obscure. I have uploaded several documents to the texts section of the [tools](#) page, as well as descriptions of these at the end of this tutorial, which will have a lot more detail on these and other techniques.

In this tutorial, we will be going over a crackme from hell that I wrote specifically for this tutorial. It shows several methods of anti-debugging. It is a very challenging crackme, and as such, this tutorial will be somewhat long and detailed. We will be comparing the source code (in assembly) along side the compiled code in Olly, so dust off that ASM book. As always, you can download all of the accompanying files from the [tutorials](#) page.

In order to make this tutorial a little less painful, I have included a picture of the entire source code, a picture of the entire disassembly (in Olly) with comments, and the assembly project for RadASM with source code, all available in the download of this tut. This way you can refer to the source and disassembly while you progress. I have also included an Olly UDD file with the disassembly fully commented. If you wish to see the crackme fully commented in Olly, just copy this file into the UDD folder in Olly, open it in a text editor, and change the file path at the top to match the path to the crackme.

My suggestion, if you would like to get the most from this tutorial, is to use a clean version of Olly with only my .ini file, meaning no plugins. Many of these techniques will still work with plugins, but this way you can see how they work without any intervention. It will also teach you about what a lot of the options in the various anti-debugging plugins are for. I will personally be using a clean install of Olly to show each technique.

Introduction

Anti-debugging techniques are methods used to fool debuggers, making the reverse engineers job harder, attempting to make the job so hard they won't want to spend the necessary time cracking the target. Some of them work on static disassemblers (like IDA Pro) and others on debuggers like Olly or SoftICE. Debuggers can be split into two types, linear sweep and recursive traversal, and some anti-debugging techniques target those specific types. Others work on all debuggers as they exploit flaws in the mechanics of debugging in general.

One of the most obvious anti-debug techniques is code obfuscation. This simply means making the code as difficult to read as possible, making the reverser's job much tougher. This includes methods such as spaghetti code (jumping all over the place), encrypting strings, making method call names meaningless (for interpreted code like VB and .NET), and code flow obfuscation where the flow of code does not follow in a linear direction.

Another type of technique is self-modifying code and polymorphism. We were gently introduced to this technique in an earlier tutorial, though these methods can become extremely complicated. This technique is used heavily in some of the more robust viruses and malware out there. Self-modifying code is a technique where the actual opcodes of the binary are changed dynamically (at run-time), making it impossible to see what the code does without stepping through it. Polymorphism is the technique of changing binary code, while still maintaining the same functionality, each time the binary is copied.

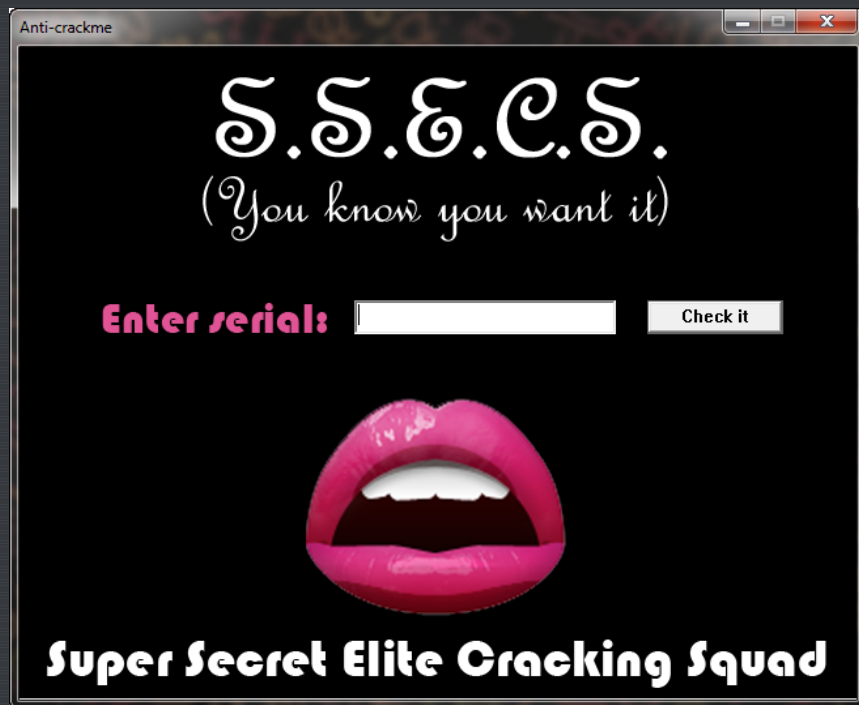
Still other techniques have to do with the way the operating system handles debugging. These include

calls to Windows API functions that tells us if the target is being debugged, checking for breakpoints dynamically in the code, removing hardware breakpoints, and using known bugs in debuggers to attempt to crash the debugger.

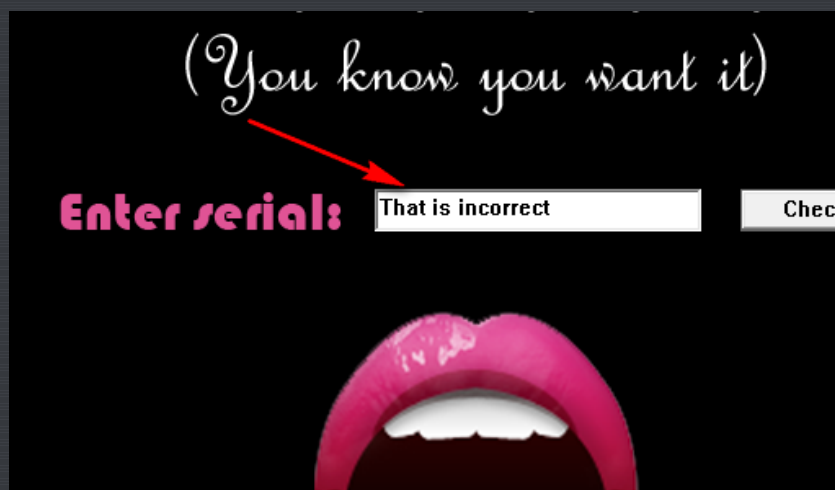
We will be discussing several of these techniques, though rest assured, there are always methods out there far more complicated than we will see here.

Introducing Our Crackme

Go ahead and run AntiCrackme1.exe outside of a debugger to see how it acts when not being debugged. Besides the anti-debugging code, it is just a simple window, asking for a serial:



It displays a goodboy or badboy depending on the serial entered:



Loading the target in Olly:

00401005	00	JMP 71000000	
00401006	00	DB 00	
0040100C	00A3 70304000	ADD BYTE PTR DS:[EBX+403070],AH	
0040100E	6A 00	PUSH 0	
0040100F	68 2B104000	PUSH Anti-cra.0040102B	
00401013	6A 00	PUSH 0	
00401015	68 1D304000	PUSH Anti-cra.0040301D	
0040101A	FF35 70304000	PUSH DWORD PTR DS:[403070]	
00401020	E8 83020000	CALL <JMP.&user32.DialogBoxParamA>	
00401025	50	PUSH EAX	
00401026	E8 A7020000	CALL <JMP.&kernel32.ExitProcess>	
0040102B	55	PUSH EBP	
0040102C	8BEC	MOV EBP,ESP	
0040102E	817D 0C 100100	CMP [ARG_2],110	
00401035	75 1A	JNZ SHORT Anti-cra.00401051	
00401037	68 B80B0000	PUSH 0BB8	
0040103C	FF75 08	PUSH [ARG_1]	
0040103F	E8 70020000	CALL <JMP.&user32.GetDlgItem>	
00401044	50	PUSH EAX	
00401045	E8 82020000	CALL <JMP.&user32.SetFocus>	
0040104A	E8 63000000	CALL Anti-cra.004010B2	
0040104F	EB 58	JMP SHORT Anti-cra.004010A9	
00401051	837D 0C 10	CMP [ARG_2],10	
00401055	75 0C	JNZ SHORT Anti-cra.00401063	
00401057	6A 00	PUSH 0	
00401059	FF75 08	PUSH [ARG_1]	
0040105C	E8 4D020000	CALL <JMP.&user32.EndDialog>	
00401061	EB 46	JMP SHORT Anti-cra.004010A9	
00401063	817D 0C 100100	CMP [ARG_2],111	
0040106A	75 34	JNZ SHORT Anti-cra.004010A0	
0040106C	8B45 10	MOV EAX,[ARG_3]	
0040106F	8B55 10	MOV EDI,[ARG_3]	
00401072	C1E9 10	SHR EDI,10	
00401075	66:0BD2	OR DX,DX	
00401078	75 2F	JNZ SHORT Anti-cra.004010A9	
0040107A	66:3D B90B	CMP AX,0BB9	
0040107E	75 1E	JNZ SHORT Anti-cra.0040109E	
00401080	8B45 08	MOV EAX,[ARG_1]	
00401083	A3 70304000	MOV DWORD PTR DS:[403070],EAX	
00401088	33C0	XOR EAX,EAX	
0040108A	A3 86304000	MOV DWORD PTR DS:[403086],EAX	
0040108F	E8 2D000000	CALL Anti-cra.004010C1	
00401094	E8 57000000	CALL Anti-cra.004010F0	
00401099	E8 72000000	CALL Anti-cra.00401110	
0040109E	EB 39	JMP SHORT Anti-cra.004010A9	
004010A0	BB	MOV EAX,0	
004010A5	C9	LEAVE	
004010A6	C2 1000	RETN 10	
004010A9	B8 01000000	MOV EAX,1	
004010AE	C9	LEAVE	
004010AF	C2 1000	RETN 10	
004010B2	E8 27020000	CALL <JMP.&kernel32.IsDebuggerPresent>	IsDebuggerPresent
004010B7	84C0	TEST AL,AL	
004010B9	75 01	JNZ SHORT Anti-cra.004010BC	

```

{Param = NULL
DlgProc = Anti-cra.0040102B
hOwner = NULL
pTemplate = "MyDialog"
hInst = NULL
DialogBoxParamA
ExitCode = 76103388
ExitProcess

```

```

ControlID = BB8 (3000.)
hwnd = 7EFDE000
GetDlgItem
hwnd = 76103388
SetFocus

```

```

Result = 0
hwnd = 7EFDE000
EndDialog

```

```

kernel32.BaseThreadInitThunk
kernel32.BaseThreadInitThunk
kernel32.BaseThreadInitThunk

```

and looking around a little, you can immediately tell that there is something different about this binary:

00401129	4F	DEC EDI	
0040112A	75 F4	JNZ SHORT Anti-cra.00401120	
0040112C	EB E7	JMP SHORT Anti-cra.00401115	
0040112E	C3	RETN	
0040112F	C9	DB 03	
00401130	03	DB 03	
00401131	00	DB 00	
00401132	00	DB 00	
00401133	00	DB 00	
00401134	3B	DB 3B	
00401135	C8	DB C8	
00401136	0F	DB 0F	
00401137	84	DB 84	
00401138	A6	DB A6	
00401139	00	DB 00	
0040113A	00	DB 00	
0040113B	00	DB 00	
0040113C	A1	DB A1	
0040113D	7C304000	DD Anti-cra.0040307C	
00401141	8B	DB 8B	
00401142	1D	DB 1D	
00401143	2A30	SUB DH,BYTE PTR DS:[EAX]	
00401145	40	INC EAX	
00401146	0038	ADD BYTE PTR DS:[EAX],BH	
00401148	D8744F E8	FIDIV DWORD PTR DS:[EDI+ECX*2-18]	
0040114C	FD	STD	
0040114D	0000	ADD BYTE PTR DS:[EAX],AL	
0040114F	00C3	ADD BL,AL	
00401151	A1 7D304000	MOV EAX,DWORD PTR DS:[40307D]	
00401156	3B08	CMP AL,BL	
00401158	74 05	JE SHORT Anti-cra.0040115F	
0040115A	E8 EE000000	CALL Anti-cra.0040124D	
0040115F	C3	RETN	
00401160	4E	DB 4E	
00401161	21	DB 21	
00401162	30	DB 30	
00401163	4C	DB 4C	
00401164	14	DB 14	
00401165	61	DB 61	
00401166	58	DB 58	
00401167	29	DB 29	
00401168	24	DB 24	
00401169	11	DB 11	
0040116A	18	DB 18	
0040116B	41	DB 41	
0040116C	4E	DB 4E	
0040116D	21	DB 21	
0040116E	B0	DB B0	
0040116F	0C	DB 0C	
00401170	25 21 58 2B 2	ASCII "%X+%F",0	
00401177	40	DB 40	
00401178	24	DB 24	
00401179	21C8	RND EAX,ECX	
0040117B	D1B9 05000000	SAR DWORD PTR DS:[ECX+5],1	
00401181	3BC8	CMP ECX,EAX	
00401183	74 5D	JE SHORT Anti-cra.004011E2	
00401185	A1 7E304000	MOV EAX,DWORD PTR DS:[40307E]	
0040118A	8B1D 05304000	MOV EBX,DWORD PTR DS:[403085]	

CHAR ':'

kernel32.BaseThreadInitThunk

CHAR 'N'
CHAR '*'
CHAR '0'
CHAR 'L'

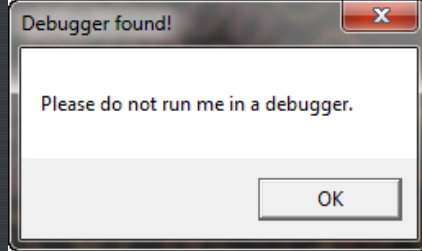
CHAR 'a'
CHAR 'X'
CHAR 'J'
CHAR '\$'

CHAR 'A'
CHAR 'N'
CHAR '*'

CHAR '@'
CHAR '\$'

kernel32.BaseThreadInitThunk

Running the crackme in Olly gives us this wonderful message:



The Beginning

Looking at the beginning of the crackme we can see that it starts with the traditional way of setting up a dialog box as the main window. If you place a breakpoint at address 40100C and then single step, as soon as you step over the call at address 401020 to DialogBoxParamA, you will notice that the entire application is contained in this call. This is because all of the code is run from callbacks based on the events of this dialog box (reminds you of Visual Basic, right?):

Everything happens here

Then we exit

In Windows, if a dialog is used as the main window, there will be a main callback that is first called, usually named "DlgProc". To find the address of this main callback, we look at the variables passed to DialogBoxParamA, and we see, in the picture above, the "DlgProc" contains a value of 40102B. This is our main DlgProc callback address.

Looking at the source code, we can see that this is a pretty straight forward affair:

```
.code
start:

    ; set up main dialog window

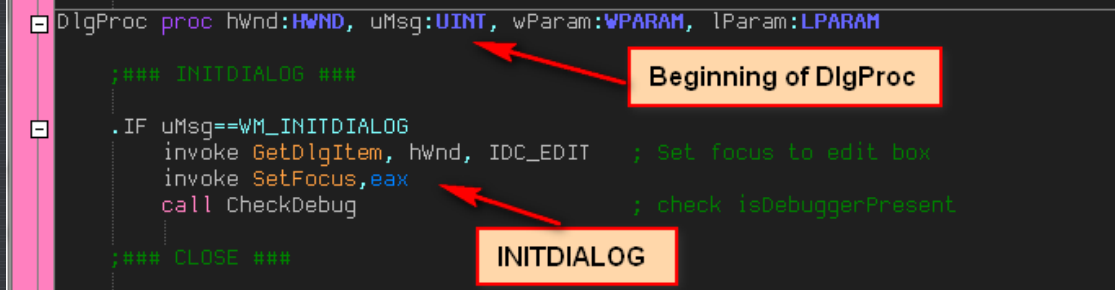
    invoke GetModuleHandle, NULL
    mov     hInstance, eax
    invoke DialogBoxParam, hInstance, ADDR DlgName, NULL, addr DlgProc, NULL
    invoke ExitProcess, eax
```

So what we want to do is place a BP at this DlgProc address and run the target, letting Windows run through till the callback gets called, and then pausing here. Place a BP at address 40102B (the address we saw above) and run Olly. We will break at the beginning of DlgProc:

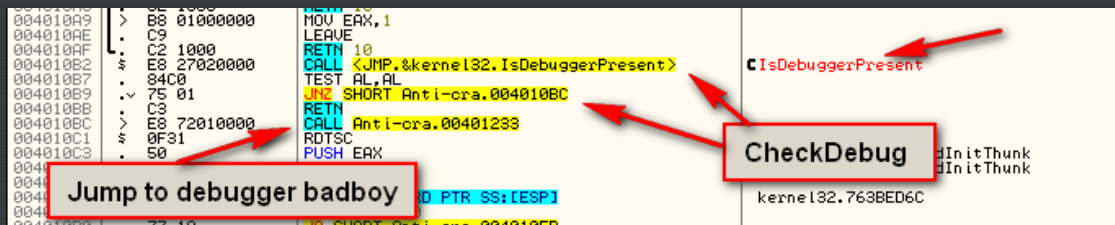
Beginning of DlgProc

We can see that, first, we compare a passed in argument with 110h, which is the ID for the Windows message WM_INITDIALOG, or the dialog initializing code. If you are a little hazy on Windows messages, please go back and re-read my [tutorial](#) on Windows messages.

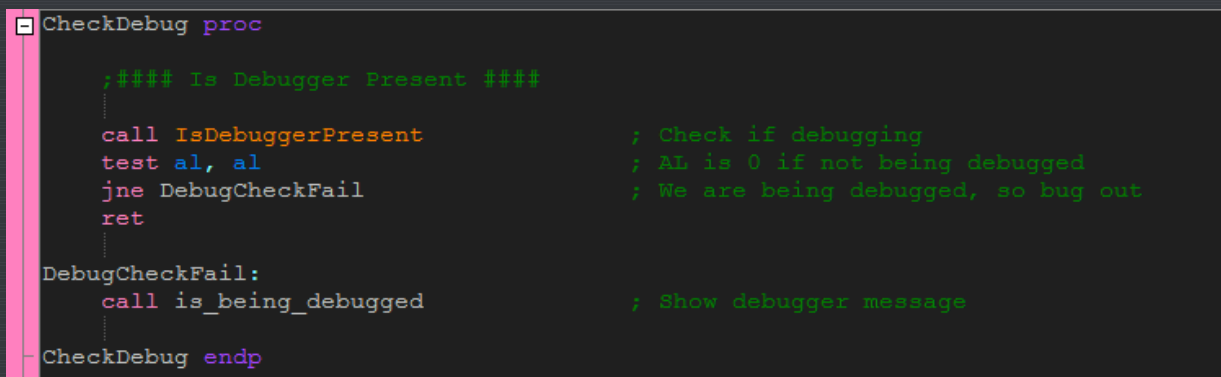
In this section there is a call to GetDlgItem and SetFocus. These just make the serial edit box have focus when the window is first loaded, so that if we start typing, the text will display in the serial box. Looking at the source code, we can see how we accomplished this:



One thing you may have noticed in the source is the call to CheckDebug. This call appears at address 40104A in the disassembly. Following this call (single-step into) we jump to the CheckDebug routine:



and following the call, we get to the CheckDebug method which includes a call to IsDebuggerPresent:



IsDebuggerPresent

One of the oldest (and most used) techniques is the IsDebuggerPresent API call, probably because it is extremely easy to use. When a binary is being debugged, Windows sets a flag in the PEB, or Process Environment Block representing this fact. The PEB is a section of code reserved with every process running on the system, setup and initialized by the Windows loader. The PEB contains information that the process needs, such as info on loaded modules, the number of processors in the system, and most importantly, a flag depicting whether the current process is being debugged or not.

A simple query to this API returns a TRUE if the current application is being debugged, and FALSE otherwise. Seeing as this is one of the most used techniques, it's also the first one used in the crackme:

*** We could also forgo calling the IsDebuggerPresent API directly, thereby removing it from the list of intermodular calls, by doing something like this:

```
MOV EAX, FS:[00000018]
```

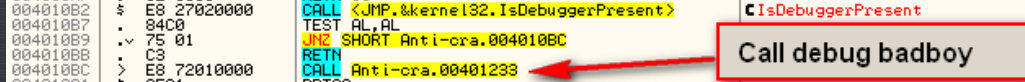
```
MOV EAX, [EAX + 0X30]
```

```
CMP BYTE PTR [EAX + 2]
```

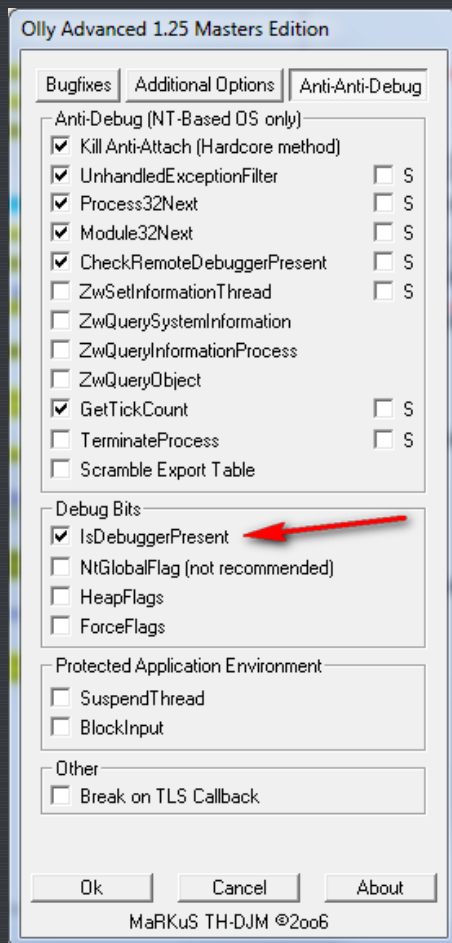
```
JNE IsBeingDebugged
```

which will load the value of the debug flag directly from the PEB, accessed through the FS register. ***

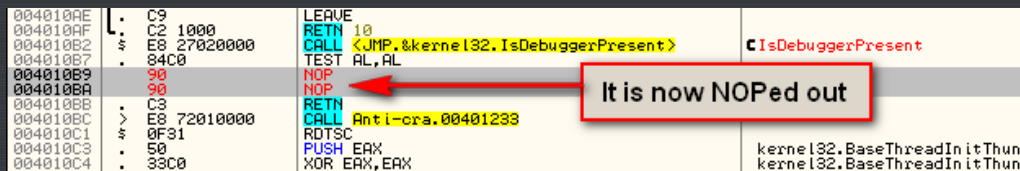
As we can see in the disassembly, we call IsDebuggerPresent, and if it returns true we jump to the call at address 4010BC, which displays the 'Found debugger' message:



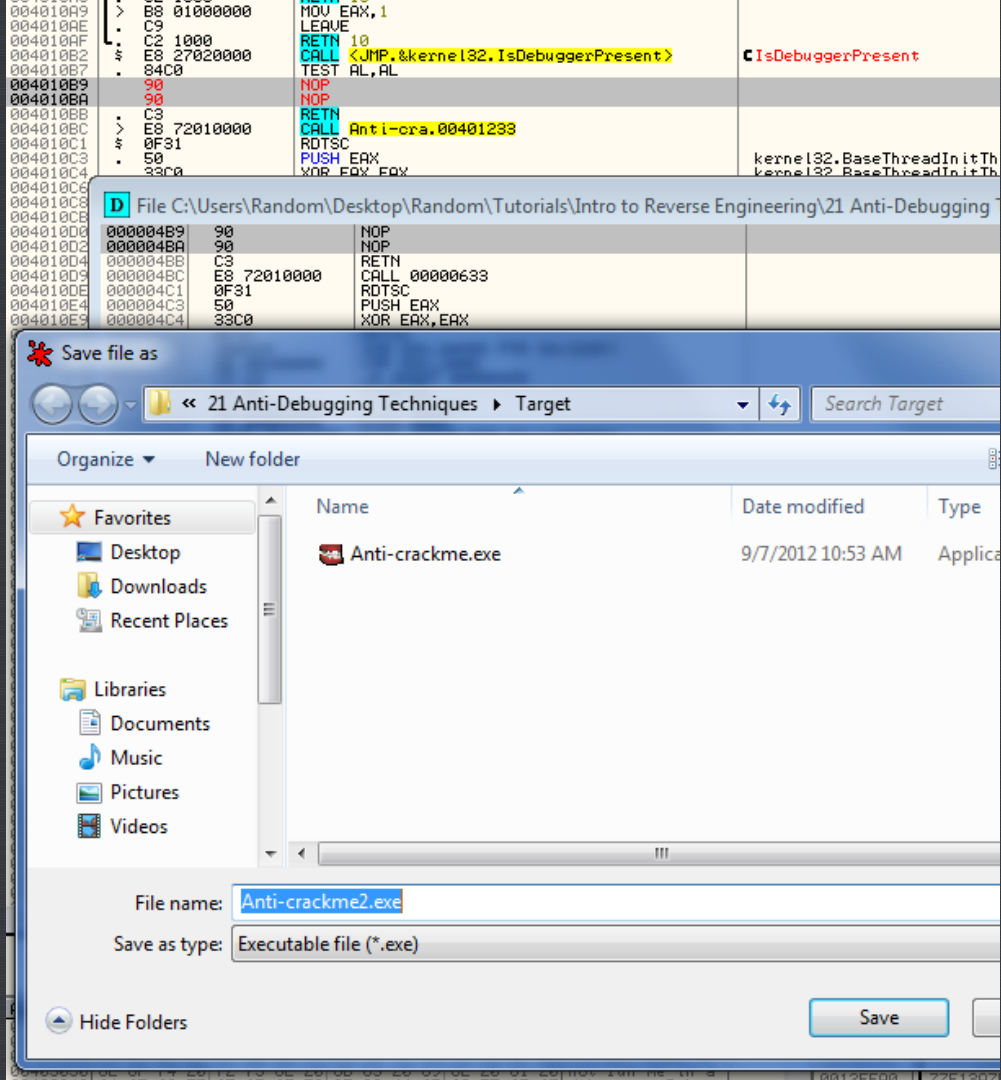
Overcoming the IsDebuggerPresent call is pretty easy in Olly as there are many plugins that do just this. The one I use is the OllyAdvanced plugin. Clicking on this (in a version of Olly that has this plugin) brings up the main settings screen. Clicking the “Anti-Anti-debug” tab, we can see where we tell OllyAdvanced to always return FALSE on a call to IsDebuggerPresent:



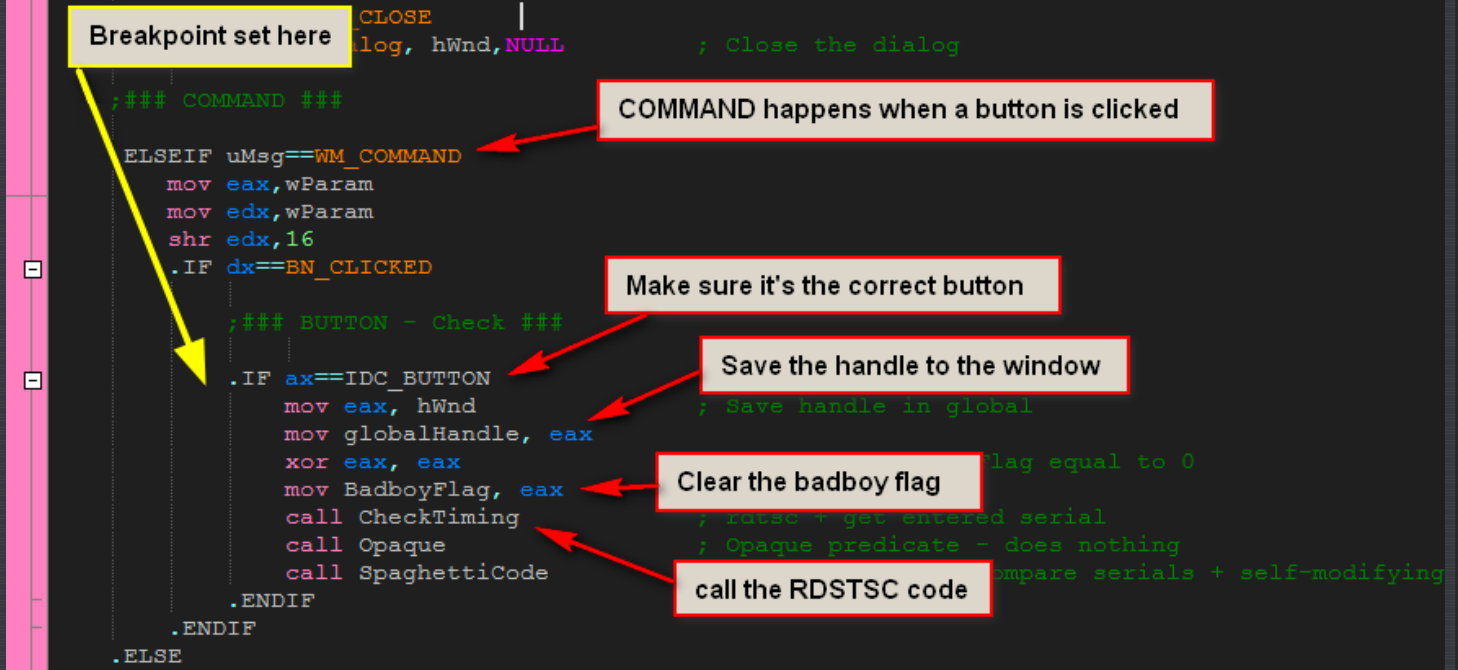
In the mean time, as we are using a ‘clean’ install of Olly, we will disable this manually by patching the check for the return result of IsDebuggerPresent at address 4010B9. Simple right-click on this instructions and select “Binary”->”Fill with NOPs”:



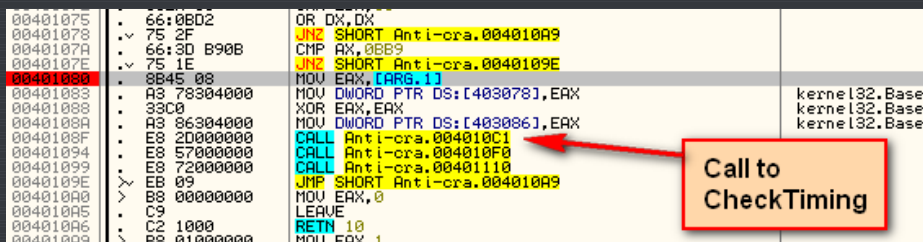
My suggestion would be to save this new version of the patched file so we don’t have to re-apply the patch every time we re-start the target. Go ahead and do that and call it Anti-crackme2.exe:



Now load this new binary in Olly and let's continue. Set a breakpoint at address 401080. This is in the main callback, `DlgProc`, but after the `INITDIALOG` message code. Putting our BP here will save us having to press F9 many times when the target is first loading as the callback will be called with every Windows message that comes through, even if we don't do anything about them. The only message we care about right now is the `WM_COMMAND` message for when our "Check it" button is pressed, and our BP is set right at the beginning of this:



When you first pressed F9, the main screen will appear, asking for a serial. Just enter any serial and click the “Check it” button. Olly will then pause at our breakpoint:



When we press the “Check it” button, a WM_COMMAND message is sent through theDlgProc callback. The first thing we do is make sure the ID matches our button’s ID, and since there’s only one button, it will. Next we save the handle to the window in a global variable so we can access it in other functions. We also reset the badboy flag back to zero (a flag used later to tell us whether the badboy should be shown or not), and then we call the CheckTiming method.

GetTickCount and RDTSC

When single stepping an executable, obviously the code does not run anywhere near as fast as if we were running the app in ‘real-time’. We can take advantage of this fact and measure how long a particular section of code takes to run. If it takes longer than a certain amount of time, we can assume that the code is being single-stepped.

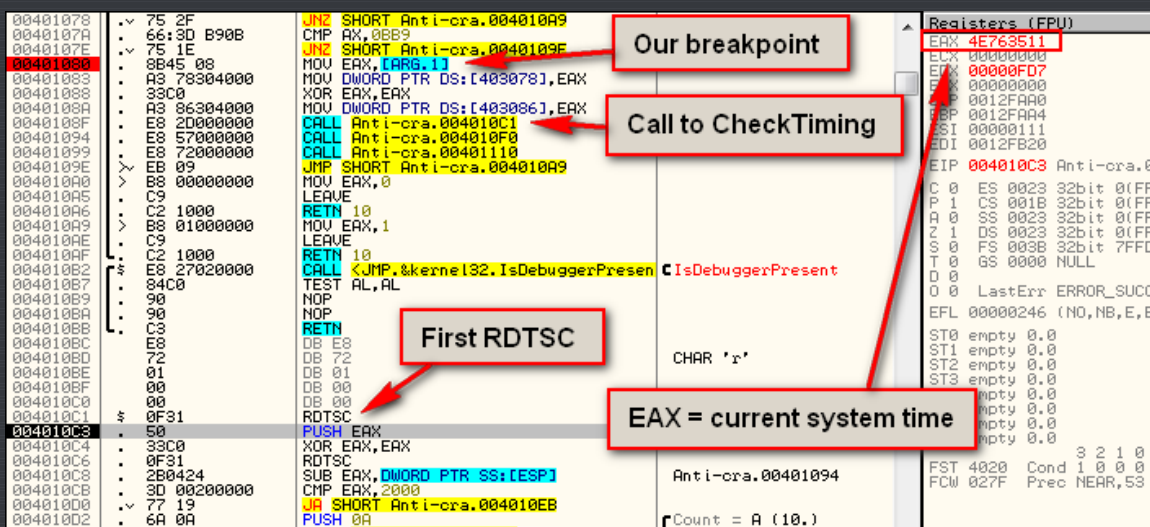
There are several ways of doing this; we could call GetTickCount before a section of code, which returns the amount of time the OS has been running, then call it again after the section of code and compare the two. If this delta is too large, we can assume that the code is being stepped.

Another technique is to use the rdtsc assembly instruction, letting the processor itself handle getting the time. Rdtsc stands for Real-Time Stamp Counter, and is a supported instruction on Intel chips. When called, EAX__ is returned containing the current amount of time the OS has been running. Again, comparing this value before and after a code section allows us to see how long it took to run that code.

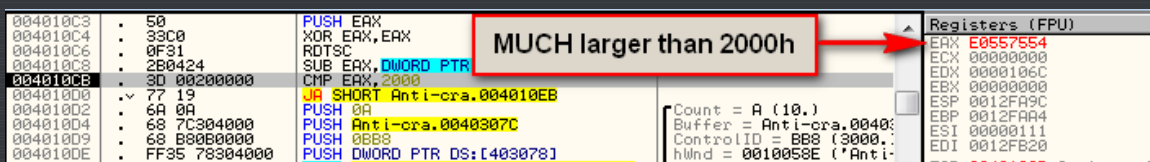
In the included crackme, we can see the use of rdtsc, though one thing needs to be pointed out; RadASM (the IDE I use) does not allow you to enter the rdtsc instruction directly, so we must insert the opcodes manually. The opcodes for this instruction are 0F 31, and in the source code you can see that I just enter them like I would raw bytes:



So, single stepping into the call at address 40108F to the CheckTiming routine, we see that first we run the RTDSC instruction which loads EAX with the current system time:



Stepping down until just past the next RTDSC instruction (to address 4010CB), we see that, after subtracting the first time from the second, the time between them was quite a bit bigger than our allotted 2000h:



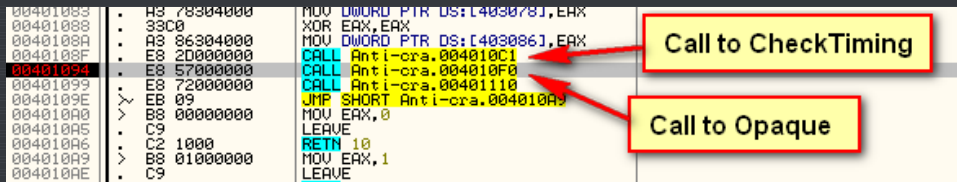
In this case, E0557554 ticks occurred between the two. Yours will be different, depending on how long you waited between the two RTDSC instructions, but it will be longer than 2000h if you single step. (You may wonder how I came up with 2000h- I did this through empirical testing. I can't guarantee that this is the best value, but it worked on all the systems I tested it on.)

There are two ways of combating this anti-debugging technique- you can NOP it out or you can simply ignore it but not single-step through this function. Either way works, but for simplicity, we will just ignore it and remember to not single step through this code. In order to get by it this first time, I just reset the zero flag at the JA instruction at address 4010D0 to bypass the target displaying the "Debug" badboy. Also, make sure that the earliest you set a breakpoint from now on is at address 401094 (the instruction right after the CheckTiming call), this way the call into the CheckTiming routine will run in real-time, bypassing this check altogether.

After the timing check, the code then calls the GetDlgItemTextA function call, which get's a pointer to the entered serial, which we will use shortly when comparing the serials:



and upon returning (to address 401094) we then immediately call into our Opaque Predicate section:

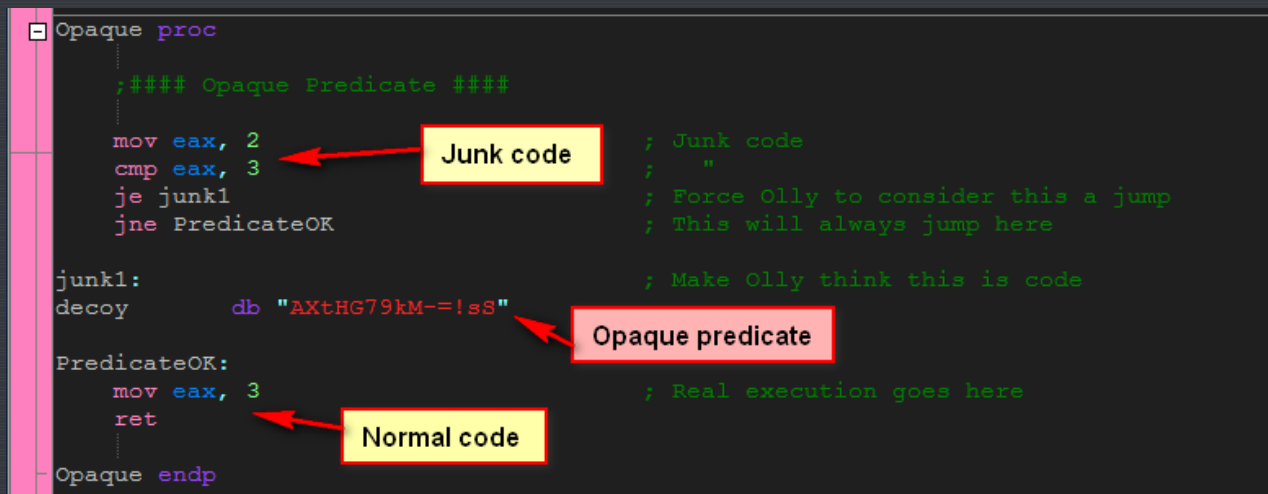


Opaque Predicates

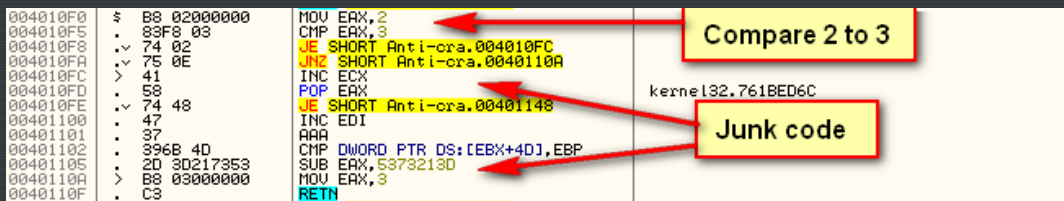
Opaque Predicates are false branches, where the branch appears to be conditional, but is not. For example, `if (1==1)` is an unconditional jump, but because of the way decompilers like Olly work, the fact that this is not really a conditional is not known.

In a normal conditional jump, there are two ways the code can go, and because of this the decompiler must disassemble the code for both conditions. In an opaque predicate, we make the disassembler think that there are two ways the code can go, even though there is really only one way it will go. The technique is to set up one of these unconditional jumps, and insert junk code into the code path that will never be called, and real code in the path that will always be called. This will force Olly to disassemble both paths, even though one of them is complete gibberish.

In our example, we see that the number '2' is compared with the number '3', which will obviously always be false. Well, obvious to us anyway. Olly is not quite as smart, and because of this, he cannot tell that we will never jump to the junk1 code, so he attempts to disassemble this code along with the PredicateOK code:



You can see that, in the path that will never be run (the junk1 section), I have simply inserted junk opcodes. I have no idea what these opcodes mean, and frankly don't care, as this will never be run. Olly, unfortunately, will attempt to disassemble these opcodes and will come up with some crazy code:



The nice thing about opaque predicates is it makes the code appear much more complicated than it really

is, making a reverse engineer spend a lot more time understanding the code. In a real application, we could even put real-looking code in the junk, sending the would-be reverser on a wild goose chase. For example, we could put a function that looks like it is comparing the entered serial with a (wrong) hard-coded serial, and then calling a goodboy or badboy depending on the results. Since this will never be run, we can put anything in here, but it will take quite a bit of digging for a reverser to figure out it's a decoy.

Spaghetti Code

The next call in our WM_COMMAND code calls the spaghetti function, the main function in our crackme:

00401080	• 8B45 08	MOV EAX,[ARG.1]	
00401083	• A3 78304000	MOV DWORD PTR DS:[403078],EAX	kernel32.BaseThreadInitThunk
00401088	• 33C0	XOR EAX,EAX	kernel32.BaseThreadInitThunk
0040108A	• A3 86304000	MOV DWORD PTR DS:[403086],EAX	kernel32.BaseThreadInitThunk
0040108F	• E8 20000000	CALL Anti-cra.004010C1	
00401094	• E8 57000000	CALL Anti-cra.004010F0	
00401099	• E8 72000000	CALL Anti-cra.00401110	
0040109E	~ EB 09	JMP SHORT Anti-cra.004010A9	
004010A0	> B8 00000000	MOV EAX,0	
004010A5	• C9	LEAVE	
004010A6	• 8B 4000	MOV EAX,[EIP+4]	

Call spaghetti code

Spaghetti code is a way to break up a normal, linear, line-by-line execution to a more frenetic, non-linear flow. For example, given the following function (in pseudo-code):

Get serial from user;

Check serial;

if(serial != hardcoded_serial)

Show badboy;

else

show goodboy;

we can change the flow to be more like this:

Jump to Spaghetti1

Spaghetti6:

Show goodboy

Jump to End

Spaghetti3:

if(serial != hardcoded_serial)

jump to Spaghetti5

else

Jump to Spaghetti6

Spaghetti1:

Get serial from user;

Jump to Spaghetti2

End:

Exit;

Spaghetti2:

Check serial

Jump to Spaghetti3

Spaghetti5:

Show badboy

Jump to End

As you can see, this is far harder to follow.

One of the initial things to notice in this code is that Olly has disassembled some of it incorrectly. In this picture, the red arrows show as data and the blue arrows are incorrect disassembly:

0040110F	.. C3	RETN	
00401110	.. \$>	E8 05000000	CALL Anti-cra.0040111A
00401115	.. \$>	E9 DF000000	JMP Anti-cra.004011F9
0040111A	.. \$>	5A	POP EDX
0040111B	.. \$>	BF 03000000	MOV EDI,3
00401120	.. \$>	8B1C8D E9114000	MOV EBX,DWORD PTR DS:[EDI*4+4011E9]
00401127	.. \$>	FFD3	CALL EBX
00401129	.. \$>	4F	DEC EDI
0040112A	.. \$>	75 F4	JNE SHORT Anti-cra.00401120
0040112C	.. \$>	EB E7	JMP SHORT Anti-cra.00401115
0040112E	.. \$>	C3	RETN
0040112F	.. \$>	B9	DB B9
00401130	.. \$>	03	DB 03
00401131	.. \$>	00	DB 00
00401132	.. \$>	00	DB 00
00401133	.. \$>	00	DB 00
00401134	.. \$>	3B	DB 3B
00401135	.. \$>	C8	DB C8
00401136	.. \$>	0F	DB 0F
00401137	.. \$>	84	DB 84
00401138	.. \$>	A6	DB A6
00401139	.. \$>	00	DB 00
0040113A	.. \$>	00	DB 00
0040113B	.. \$>	00	DB 00
0040113C	.. \$>	A1	DB A1
0040113D	.. \$>	7C304000	DD Anti-cra.0040307C
00401141	.. \$>	8B	DB 8B
00401142	.. \$>	1D	DB 1D
00401143	.. \$>	2A30	SUB DH,BYTE PTR DS:[EAX]
00401145	.. \$>	40	INC EAX
00401146	.. \$>	0038	ADD BYTE PTR DS:[EAX],BH
00401148	.. \$>	D8744F E8	FDIV DWORD PTR DS:[EDI+ECX*2-18]
0040114C	.. \$>	FD	STD
0040114D	.. \$>	0000	ADD BYTE PTR DS:[EAX],AL
0040114F	.. \$>	00C3	ADD BL,AL
00401151	.. \$>	A1 7D304000	MOV EAX,DWORD PTR DS:[40307D]
00401156	.. \$>	3BD8	CMF AL,BL
00401158	.. \$>	74 05	JE SHORT Anti-cra.0040115F
0040115A	.. \$>	E8 EE000000	CALL Anti-cra.0040124D
0040115F	.. \$>	C3	RETN

kernel32.761BED6C

CHAR ','

kernel32.BaseThread

This code is not correct

Going to the source code, we can see why Olly has gotten confused:

```
Loop1:
    mov ebx, dword ptr [edi*4+address_array] ; Get address to execute
    call ebx ; Execute part of function
    dec edi ; Do it three times
    jnz Loop1
    jmp Return
    ret

code3:
    mov ecx, 3
    cmp ecx, eax
    je junk2

    ; ### Compare first two digits of serials ###

    mov eax, dword ptr buffer ; Get first character of entered
    mov ebx, dword ptr DebuggerText+4 ; Get the "c" in DebuggerText
    cmp al, bl ; Are they the same?
    je Compare3 ; yes, so keep going
    call ShowBadboy
    ret

Compare1:
    mov eax, dword ptr buffer+1 ; Get first character of entered
    cmp al, bl ; Are they the same?
    je Compare2 ; yes, so keep going
    call ShowBadboy

Compare2:
    ret
```

This will always jump

So what is this return for?

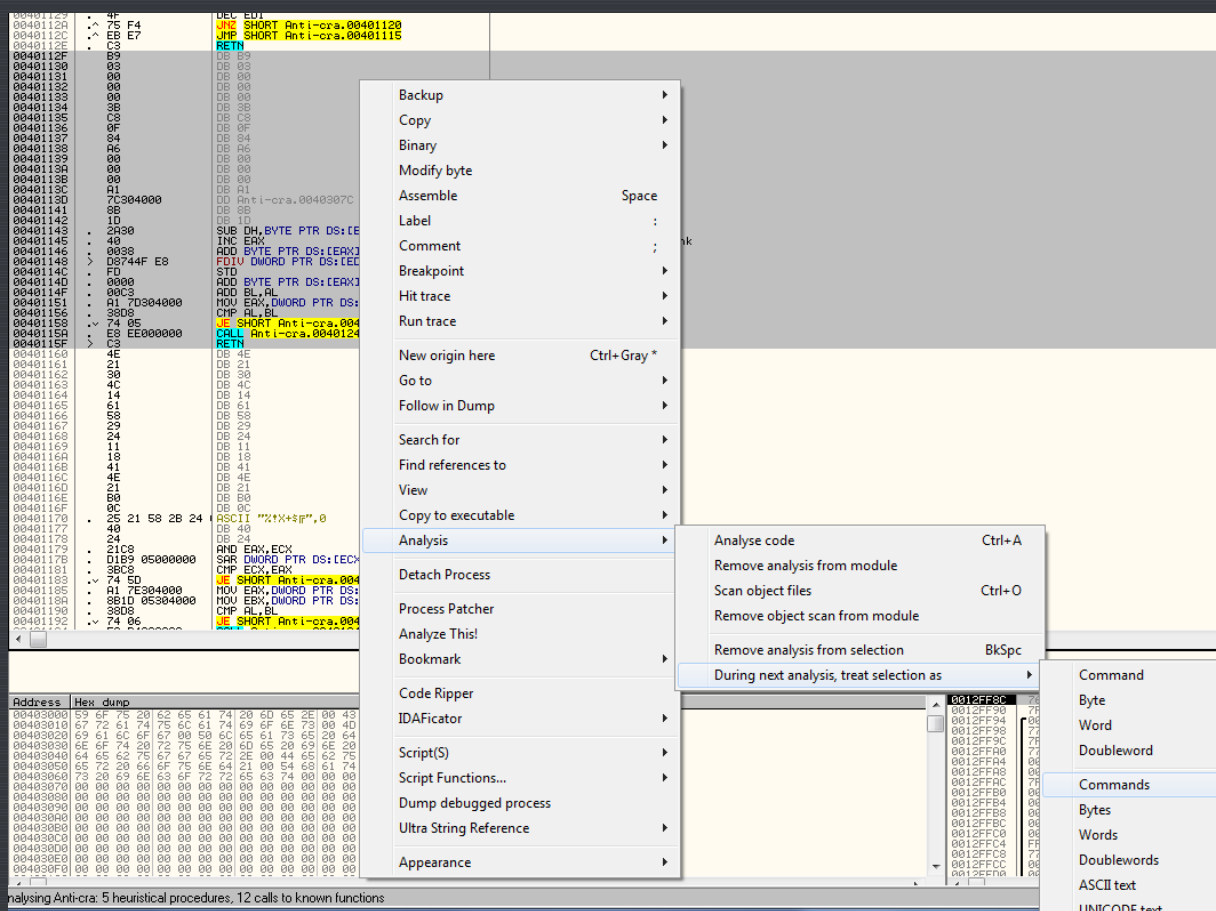
All of this code will decompile incorrectly

Junk code 1

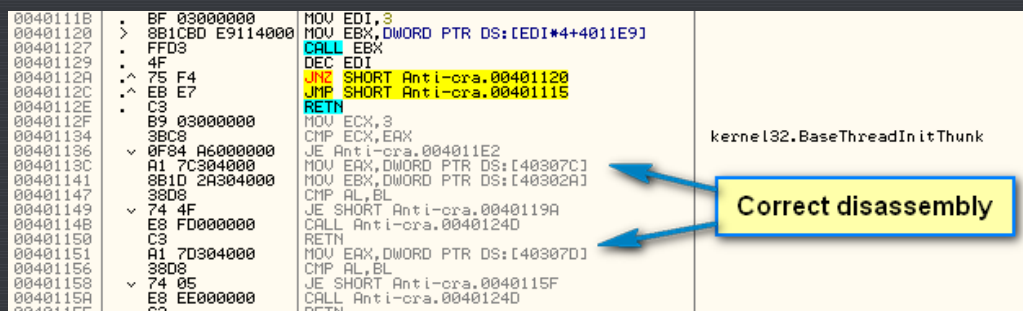
Until we get to the next return

We can see that something similar to an opaque predicate has been used- it is a jump followed by a

return. What is Olly to make of this? Well, it resorts to displaying until the next return as data. We have to tell Olly that it is in fact instruction opcodes and not data. The way to do this is to highlight all of the incorrectly decompiled code, right-click and choose "Analysis"->"During next analysis, treat selection as"->"Commands":



Now Olly will re-analyze this section, assuming that these lines are instructions and not data. Now Olly shows the correct disassembly:



That will help as we go through the code!

The Table Interpretation Method

In addition to mixing up the flow of the code, this crackme also utilizes a technique called table Interpretation (there are actually additional techniques to this, but I am just showing the most basic way). This further obfuscates the code in that it's a lot harder to figure out where the execution will jump to next. It works by loading several addresses into an array. These addresses are entry points into sections of code. We then enter a loop that loads each address and calls each one in order. These code sections are usually compiled out of order, making it harder to follow the execution.

The 'normal' way we would call each section would be something like this:

call code1

call code2

call code3

But in the crackme, the way we call each section is more like this (in psuedo-code):

```
array = [ 0, address of code 3, address of code 2, address of code 1];
```

```
for( i = 3; i > 0; i-)
```

```
{
```

```
call array[i];
```

```
}
```

This adds a level of obfuscation in that we are calling each section indirectly.

Here we can see the loop that loads each address and calls them, one after another:

```
##### Spaghetti Code #####

call Spaghetti                                ; Push this address on to the stack

Return:
jmp Continue                                ; Done with cpaghetti code

Spaghetti:
pop edx                                     ; Store return value in edx
mov edi, 3                                ; index into call order (in array of addresses)

Loop1:
mov ebx, dword ptr [edi*4+address_array]    ; Get address to execute
call ebx                                    ; Execute part of function
dec edi                                    ; Do it three times
jnz Loop1
jmp Return
ret

code3:
mov ecx, 3                                ; Junk code 1
cmp ecx, eax                               ; "
je junk2                                   ; "

; ### Compare first two digits of serials ###
```

Load next address in array

and jump to it

and here's the actual array that holds the addresses:

```
Compare6:
ret

junk2:
invoke ExitProcess,0                        ; This will never be reached

address_array dd 0, code3, code2, code1
```

The addresses we will jump to

Another benefit to this is that it inserts data (the array of addresses) into the middle of code, making it harder for the disassembler to disassemble the code:

00401115	E9 DF000000	JMP Anti-cra.004011F9	
0040111A	5A	POP EDX	
0040111B	BF 03000000	MOV EDI,3	
00401120	> 8B1CB0 E9114000	MOV EBX,DWORD PTR DS:[EDI*4+4011E9]	
00401127	FFD3	CALL EBX	
00401129	4F	DEC EDI	
0040112A	75 F4	JNZ SHORT Anti-cra.00401120	
0040112C	EB E7	JMP SHORT Anti-cra.00401115	
0040112E	C3	RETN	
0040112F	B9 03000000	MOV ECX,3	
00401134	3BC8	CMP ECX,EAX	
00401136	> 0F84 A6000000	JE Anti-cra.004011E2	
0040113C	A1 7C304000	MOV EAX,DWORD PTR DS:[40307C]	
00401141	8B1D 2A304000	MOV EBX,DWORD PTR DS:[40302A]	
00401147	3BD8	CMP AL,BL	
00401149	? 74 4F	JE SHORT Anti-cra.0040119A	
0040114B	? E8 FD000000	CALL Anti-cra.0040124D	
00401150	? C3	RETN	

This is the actual array

The calls start with the last address and work their way down, so our calls will happen in this order; code1, code2, code3. As we can see, these code sections have been compiled in reverse order, making it more difficult to follow code execution:

```

SpaghettiCode proc
    ;==== Spaghetti Code ====
    call Spaghetti                ; Push this address onto the stack
Return:
    jmp Continue                ; Done with spaghetti code
Spaghetti:
    pop edx                    ; Store return value in edx
    mov edi, 3                ; Index into call under (in array of addresses below)
Loop1:
    mov ebx, dword ptr [edi*4+address_array] ; Get address to execute
    call ebx                  ; Execute part of function
    dec edi                    ; Do it three times
    jnz Loop1
    jmp Return
    ret

code3:
    mov ecx, 3                ; Junk code 1
    cmp ecx, eax
    je junk2

    ; === Compare first two digits of serials ===
    mov eax, dword ptr buffer ; Get first character of entered serial
    mov ebx, dword ptr DebuggerText+4 ; Get the "c" in DebuggerText test
    cmp al, bl
    je Compare3
    call ShowBadboy
    ret

Compare1:
    mov eax, dword ptr buffer+1 ; Get first character of entered serial
    cmp al, bl
    je Compare2
    call ShowBadboy

Compare2:
    ret

;==== Goodbye Message Box Code =====
Goodbye:
    ; Real code in middle of junk
    db 4eh, 21h, 30h, 4ch, 14h, 61h, 58h, 29h, 24h, 11h, 18h, 41h, 4eh, 21h,
    80h, 0ch, 25h, 21h, 58h, 2bh, 24h, 0c9h, 80h, 40h, 24h, 21h, 0c8h, 0d1h

    ; invoke MessageBoxW, MB_OK, addr GoodbyeText, addr GoodbyeCaption, MB_OK
    ; invoke ExitProcess,0
    ; nop
    ; nop

;=====
code2:
    ; Junk code
    mov ecx, 5
    cmp ecx, eax
    je junk2

    ; === Compare third and fourth digits of serials ===
    mov eax, dword ptr buffer+2 ; Get third character of entered serial
    mov ebx, dword ptr GoodbyeText+5 ; Get the "e" in GoodbyeText
    cmp al, bl
    je Compare3
    call ShowBadboy
    ret

Compare3:
    mov eax, dword ptr buffer+3 ; Get first character of entered serial
    mov ebx, dword ptr BadboyText+10 ; Get the "c" in Badboy Text
    cmp al, bl
    je Compare4
    call ShowBadboy

Compare4:
    ret

code1:
    ; Junk code
    mov ecx, 0
    cmp ecx, eax
    je junk2

    ; === Compare fifth and last digits of serials ===
    mov eax, dword ptr buffer+4 ; Get third character of entered serial
    mov ebx, dword ptr BadboyText+6 ; Get the "e" in BadboyText
    cmp al, bl
    je Compare5
    call ShowBadboy
    ret

Compare5:
    mov eax, dword ptr buffer+5 ; Get first character of entered serial
    mov ebx, dword ptr DebugCaption+14 ; Get the "r" in DebugCaption
    cmp al, bl
    je Compare6
    call ShowBadboy

Compare6:
    ret

```

Also, junk code has been inserted into the beginning of each code section to lower readability .


Checking Against Pre-Existing Strings

In each code section, a part of the actual pertinent code is run. In the first section (code1) we check the first two characters entered against the first two characters of our hard-coded serial. There is a further obfuscation performed here, though. In order to not have a hardcoded serial (which can be pretty easily detected in Olly), this crackme checks the characters entered against characters from strings that have nothing to do with the serial. For example, the first entered character is compared to the fifth character of DebuggerText:

```
; ### Compare first two digits of serials ###
mov  eax, dword ptr buffer          ; Get first character of entered serial
mov  ebx, dword ptr DebuggerText+4 ; Get the "c" in DebuggerText text
cmp  al, bl                         ; Are they the same?
je   Compare3                      ; yes, so keep going
call ShowBadboy
ret

Compare1:
mov  eax, dword ptr buffer+1        ; Get second character of entered serial
cmp  al, bl                         ; Are they the same?
je   Compare2                      ; yes, so keep going
call ShowBadboy
ret

Compare2:
ret
```




Fifth character from DebuggerText

and here is the letter it compares it to:

```
.data
GoodboyText      db  "You beat me.",0
GoodboyCaption   db  "Congratulations", 0
DlgName          db  "MyDialog",0
DebuggerText     db  "Please do not run me in a debugger.",0

DebugCaption     db  "Debugger found!",0
BadboyText       db  "That is incorrect",0
```



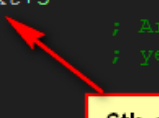
So we can see that the first character of the entered serial is compared with 's'. Right below this check in the source code, we check the second character of the entered serial with this same 's'. So we know the first two characters of the serial should be 'ss'.

The next compare section checks the 3rd and 4th characters entered with characters from GoodboyText and BadboyText:


```
; ### Compare third and fourth digits of serials ###
mov  eax, dword ptr buffer+2        ; Get third character of entered serial
mov  ebx, dword ptr GoodboyText+5   ; Get the "e" in GoodboyText
cmp  al, bl                         ; Are they the same?
je   Compare3                      ; yes, so keep going
call ShowBadboy
ret

Compare3:
mov  eax, dword ptr buffer+3        ; Get first character of entered serial
mov  ebx, dword ptr BadboyText+10   ; Get the "c" in Badboy text
cmp  al, bl                         ; Are they the same?
je   Compare4                      ; yes, so keep going
call ShowBadboy
ret

Compare4:
ret
```



6th character of GoodboyText



11th character of BadboyText

and here's the strings:

```
.data
GoodboyText      db  "You beat me.",0
                  ↑
GoodboyCaption   db  "Congratulations", 0
DlgName          db  "MyDialog",0
DebuggerText     db  "Please do not run me in a debugger.",0
DebugCaption     db  "Debugger found!",0
BadboyText       db  "That is incorrect",0
                  ↑
```

So we know the third character of the entered text should be 'e' and the fourth should be 'c'. Finally we check the fifth and last characters:

```
; ### Compare fifth and last digits of serials ###

mov eax, dword ptr buffer+4      ; Get third character of entered serial
mov ebx, dword ptr BadboyText+6  ; Get the "s" in BadboyText
cmp al, bl                       ; Are they the same?
je Compare5                      ; yes, so keep going
call ShowBadboy
ret

Compare5:
mov eax, dword ptr buffer+5      ; Get first character of entered serial
mov ebx, dword ptr DebugCaption+14 ; Get the "!" in DebugCaption
cmp al, bl                       ; Are they the same?
je Compare6                      ; yes, so keep going
call ShowBadboy

Compare6:
ret
```

to the appropriate strings:

```
DebugCaption     db  "Debugger found!",0
                  ↑
BadboyText       db  "That is incorrect",0
                  ↑
.data?
```

and we see that the entire serial is 'ssecs!'.

Searching for breakpoints

After the spaghetti code, the crackme does not return control to the main WM_COMMAND section, instead it stays in this call, as a way to further obfuscate the code. Right after Spaghetti is done, we jump to the Continue label, as we can see here in the source code:

```

SpaghettiCode proc
;#### Spaghetti Code ####
call Spaghetti
Return:
jmp Continue
Spaghetti:
pop edx
mov edi, 3

```

When spaghetti is done...

...we jump to continue

At the beginning of Continue, we immediately call the BreakpointCheck code. After this, we fall through to checking if the badboy flag was set, and if it wasn't, we fall through to the self-modifying code section:

```

Continue:
call BreakpointCheck
;#### Check badboy flag ####
mov eax, offset BadboyFlag
mov eax, [eax]
cmp eax, 0
je BeginShowGoodboy
ret
;#### Self-Modifying Code ####
BeginShowGoodboy:
mov edi, Goodboy
mov ecx, 7
mov eax, "AX!$"

```

First we call BreakpointCheck

Then check the state of the badboy flag

And start decrypting data if it's not set

Following is the BreakpointCheck routine. The purpose of this code is to search the entire contents of our application (in memory), looking for a breakpoint. If we find one, we know we're being debugged and we show the debugging message:

```

BreakpointCheck proc
;#### BP Checking ####
mov eax, start
mov ecx, ProgramEnd
sub ecx, start
ProgramEnd:
nop
Loop3:
cmp byte ptr [eax], 0CCh
jz is_being_debugged
inc eax
dec ecx
jnz Loop3
ret
BreakpointCheck endp

```

Scan entire code for breakpoints

Jump to is_being_debugged if we find a BP (0xCC)

As we will see at the end of this tutorial (in the checksums section), we will go into more detail of this process. But in the meantime, we need to know when setting a breakpoint, the memory contents of the first byte of the instruction where we place the breakpoint is replaced by the constant CCh. This is the opcode for an interrupt, and Olly is registered to trap this specific breakpoint. This way, as the program is executing in memory, when an interrupt occurs execution is given to Olly by the operating system, and Olly will pause the execution of the executable here, giving us control of the target.

*** Please see the checkpoints section at the end for a more detailed explanation of breakpoints and interrupts. **

This routine searches our code looking for the tell-tale sign of a breakpoint. If we encounter a CCh anywhere in this code, we break and display the badboy. The beginning address of our program is defined by "start", the label placed at the top of our source code. To find the length of our code, we subtract the address of the beginning of from the address of the end, which gives us the length. Next we search every byte, comparing each with the constant CCh. If CCh is not found, we simply return from the procedure.

Here's what the breakpoint check looks like in Olly:

0040127C	. 68 B0B00000	PUSH 0BB8	ControlID = BB8 (3000.)
00401281	. FF35 78304000	PUSH DWORD PTR DS:[403078]	hWnd = NULL
00401287	. E8 3A000000	CALL <JMP.&user32.SetDlgItemTextA>	SetDlgItemTextA
0040128C	. C3	RETN	
0040128D	. \$ B8 00104000	MOV EAX, Anti-cra.<ModuleEntryPoint>	
00401292	. B9 9D124000	MOV ECX, Anti-cra.0040129D	
00401297	. 81E9 00104000	SUB ECX, Anti-cra.<ModuleEntryPoint>	
0040129D	. 90	NOP	
0040129E	. > 8038 CC	CMP BYTE PTR DS:[EAX],0CC	
004012A1	. ^ 74 90	JE SHORT Anti-cra.004012B3	
004012A3	. . 40	INC EAX	kernel32.BaseThreadInitThunk
004012A4	. . 49	DEC ECX	
004012A5	. ^ 75 F7	JNZ SHORT Anti-cra.0040129E	
004012A7	. C3	RETN	
004012A8	. \$- FF25 28204000	JMP DWORD PTR DS:[<&user32.DialogBoxParamA	user32.DialogBoxParamA
004012AE	. \$- FF25 1C204000	JMP DWORD PTR DS:[<&user32.EndDialog>]	user32.EndDialog
004012B4	. \$- FF25 18204000	JMP DWORD PTR DS:[<&user32.SetDlgItemTextA	user32.SetDlgItemTextA

In order to overcome this anti-debugging technique, we simply need to NOP out the call to it. After a little searching, we find the call at address 4011F9:

004011E4	. E8 E9000000	CALL <JMP.&kernel32.ExitProcess>	ExitProcess
004011E9	. . 00000000	DD 00000000	
004011ED	. 2F114000	DD Anti-cra.0040112F	
004011F1	. 7C114000	DD Anti-cra.0040117C	
004011F5	. AF114000	DD Anti-cra.004011AF	
004011F9	. > E8 8F000000	CALL Anti-cra.0040128D	
004011FE	. B8 86304000	MOV EAX, Anti-cra.00403086	
00401203	. . 8B00	MOV EAX, DWORD PTR DS:[EAX]	
00401205	. . 83F8 00	CMP EAX, 0	
00401208	. ^ 74 01	JE SHORT Anti-cra.0040120B	
0040120A	. C3	RETN	

Highlighting this instruction, right-click and select "Modify"->"Fill with NOPs". This replaces the instruction with NOP opcodes:

004011E9	. . 00000000	DD 00000000	
004011ED	. 2F114000	DD Anti-cra.0040112F	
004011F1	. 7C114000	DD Anti-cra.0040117C	
004011F5	. AF114000	DD Anti-cra.004011AF	
004011F9	. 90	NOP	
004011FA	. 90	NOP	
004011FB	. 90	NOP	
004011FC	. 90	NOP	
004011FD	. 90	NOP	
004011FE	. . B8 86304000	MOV EAX, Anti-cra.00403086	
00401203	. . 8B00	MOV EAX, DWORD PTR DS:[EAX]	
00401205	. . 83F8 00	CMP EAX, 0	
00401208	. ^ 74 01	JE SHORT Anti-cra.0040120B	
0040120A	. C3	RETN	
0040120B	. \$- FF60114000	CALL Anti-cra.00401160	

Now we have no more breakpoint check. At this point, I would again save the binary (Anti-crackme3.exe) so we don't have to remember to apply the patch every time we run the target.

Self-Modifying Code

The final anti-debugging technique used in this crackme is self-modifying code. This is a technique whereby the opcodes in the binary get changed at run-time into a different set of opcodes. In the case of this crackme, it changes a bunch of random data into a call to show the goodboy. There are plenty of reasons this technique is used: the function call encrypted in this section will not show up in the intermodular calls, the random data can trick disassemblers into thinking its code, and after the opcodes get decrypted, you must tell Olly to re-analyze these bytes as opcodes instead of data.

There are many more facets of self-modifying code than I have displayed here, some of them next to impossible to work with (but not completely impossible....why? cause that's impossible...)

If you look in the middle of the spaghetti code in Olly, you will notice this random data looks somewhat out of place in the middle of code:

00401158	· 74 05	JE SHORT Anti-cra.0040115F	
0040115A	· E8 EE000000	CALL Anti-cra.0040124D	
0040115F	> C3	RETN	
00401160		DB 4E	CHAR 'N'
00401161		DB 21	CHAR 't'
00401162		DB 30	CHAR '0'
00401163		DB 4C	CHAR 'L'
00401164		DB 14	
00401165		DB 61	CHAR 'a'
00401166		DB 58	CHAR 'x'
00401167		DB 29	CHAR 'j'
00401168		DB 24	CHAR '\$'
00401169		DB 11	
0040116A		DB 18	
0040116B		DB 41	CHAR 'A'
0040116C		DB 4E	CHAR 'N'
0040116D		DB 21	CHAR 't'
0040116E		DB B0	
0040116F		DB 0C	
00401170	· 25 21 58 2B 24	ASCII "%!X+\$f",0	
00401177		DB 40	CHAR '@'
00401178		DB 24	CHAR '\$'
00401179	· 21C8	AND EAX,ECX	
0040117B	· 01B3 05000000	SAR DWORD PTR DS:[ECX+5],1	
00401181	· 3BC8	CMP ECX,EAX	
00401183	· 74 5D	JE SHORT Anti-cra.004011E2	kernel32.BaseTh

What is this?

*** Because this really is raw data (for now), if we tell Olly to re-analyze this as commands, it will be complete gibberish. ***

Here is the source code for this section. Notice that it defines data in the middle of the code section, and our compiler is more than happy to insert the data amongst the code. Also, I have added comments under the code as to what these bytes will represent once they've been decrypted:

```

mov eax, dword ptr buffer+1 ; Get first character of entered serial
cmp al, b1 ; Are they the same?
je Compare2 ; yes, so keep going
call ShowBadboy

Compare2:
ret

;#### Goodboy Message Box Code ####
Goodboy: ; Real code in middle of junk

db 4eh, 21h, 30h, 4ch, 14h, 61h, 58h, 29h, 24h, 11h, 18h, 41h, 4eh, 21h,
0b0h, 0ch, 25h, 21h, 58h, 2bh, 24h, 0c9h, 00h, 40h, 24h, 21h, 0c8h, 0d1h

; invoke MessageBoxA, NULL, addr GoodboyText, addr GoodboyCaption, MB_OK
; invoke ExitProcess,0
; nop
; nop

;#####

code2: ; Junk code
mov ecx, 5 ; "
cmp ecx, eax ; "
je junk2 ; "

```

Code

Data in the middle of code

This is what was encrypted to get this data

Code

In the case of this crackme, the original instructions were simply XORed with a certain key in order to get the raw data. Because they were XORed, XORing the encrypted data with the same key will decrypt them back to their original.

*** Many commercial software titles would do far more than XOR this code to decrypt it, but you'd be surprised how many use simple algorithms. ***

Now, let's take a look at the code that does the decrypting:

```

;#### Self-Modifying Code ####

BeginShowGoodboy:
    mov edi, Goodboy
    mov ecx, 7
    mov eax, "AX!$"

Loop2:
    xor [edi], eax
    add edi, 4
    dec ecx
    jne Loop2

```

Get beginning address of encrypted data

Loop 7 times ; Length of goodboy code ; XOR value

and XOR each section with AX!\$

XOR encrypted data, saving back into binary

and jump to next section of data

Here, we load the address of the beginning of our encrypted data and load ECX with 7 which is the number of times we will loop through this code. The reason it's seven is that we will XOR 4 bytes at a time, XORing the 4 bytes with "AX!\$". 7 times 4 equals 28, so we will XOR 28 bytes. You may notice that the call to MessageBox and ExitProcess are only 26 bytes, so I added two NOPs at the end so we didn't start decrypting the next section of code.

Next we do the actual XORing of the bytes, cycling through all of our data. If you place a breakpoint at address 40120B (the beginning of the decryption routine) and hit F9, each time you run the app you will see a section of code, starting at address 401160, change into decrypted code. Of course, Olly doesn't know that these changed bytes are instructions, so it tries to disassemble after each pass of the loop. After seven times through the loop, you will see the decrypted bytes in their entirety (keep in mind that you will only see this if you enter the correct serial):

0040115A	> E8 EE000000	CALL Anti-cra.00401240	
0040115F	C3	RETN	
00401160	6A 00	DB 6A	CHAR 'j'
00401161	68 00	DB 00	CHAR 'h'
00401162	68 00	DB 68	CHAR '0'
00401163	00 30	DB 00	CHAR '@'
00401164	00 40	DB 30	CHAR 'h'
00401165	00 00	DB 40	CHAR '@'
00401166	68 00	DB 00	CHAR 'j'
00401167	00 00	DB 68	CHAR 'h'
00401168	30 00	DB 00	CHAR '0'
00401169	40 00	DB 30	CHAR '@'
0040116A	00 00	DB 40	CHAR '@'
0040116B	6A 00	DB 00	CHAR 'j'
0040116C	00 00	DB 6A	CHAR 'h'
0040116D	00 00	DB 00	CHAR '0'
0040116E	E8 40	DB E8	CHAR '@'
0040116F	40 00	DB 40	CHAR 'M'
00401170	01 00 00 6A 00	ASCII "0"	
00401171	01 00	DB 01	
00401172	00 00	DB 00	
00401173	00 90 90 90 50	ADD BYTE PTR DS:[EAX+5B9901],DL	
00401174	00 00	ADD BYTE PTR DS:[EAX],AL	
00401175	3BC8	CMP ECX,EAX	
00401176	74 5D	JE SHORT Anti-cra.004011E2	
00401177	A1 7E304000	MOV EAX,DWORD PTR DS:[40307E]	
00401178	8B D0	MOV EBX,DWORD PTR DS:[40307E]	

Let's tell Olly that this is code and not data. Select all the modified lines, right-click and choose "Analysis"->"During next analysis, treat selection as"->"Commands" and we will see the decrypted code as it's meant to be seen:

0040115A	> E8 EE000000	CALL Anti-cra.00401240	
0040115F	C3	RETN	
00401160	6A 00	PUSH 0	
00401161	68 00304000	PUSH Anti-cra.00403000	
00401162	68 00304000	PUSH Anti-cra.00403000	ASCII "Congratulations"
00401163	6A 00	PUSH 0	ASCII "You beat me."
00401164	E8 40010000	CALL <JMP.&user32.MessageBoxA>	
00401165	6A 00	PUSH 0	
00401166	E8 58010000	CALL <JMP.&kernel32.ExitProcess>	
00401167	90	NOP	
00401168	90	NOP	
00401169	B9 05000000	MOV ECX,5	
0040116A	3BC8	CMP ECX,EAX	
0040116B	74 5D	JE SHORT Anti-cra.004011E2	
0040116C	A1 7E304000	MOV EAX,DWORD PTR DS:[40307E]	

We can now see what's going on!

*** By the way, if you wish to experiment with self-modifying code on your own, there are a couple of additional things you need to know. At the end of this tutorial, I have included a section on changing PE file section characteristics, so read this before attempting this technique on your own. ***

After decrypting the code, there is one more little surprise we have to deal with...

Return Obfuscation

You can see in the source code that the crackme does not just simply jump to the decrypted code here, but performs some strange computations. This is a simple anti-debugging technique that just makes reverse engineering the code a little harder:

```

;#### Return Obfuscation ####

push (Goodboy + 754841h)           ; Push return value + constant
mov eax, [esp]
sub eax, 754841h                   ; and subtract back the constant
mov [esp], eax
ret

```

SpaghettiCode endp

A simple version of return obfuscation is something like this:

push offset code_to_call

ret

What it does is changes a jump into a return. This code is equivalent to:

jump code_to_call

but has the benefit of being a little more complicated in the code, not to mention making Olly's job harder. Where this technique really shines is in static disassembly, using something like IDA Pro. Because there is a return instead of a jump, it's a lot harder for IDA, as well as the reverser, to figure out which call this return goes to.

The reason why these two snippets do the same thing is that when the CPU performs a return, it pulls the return address from the top of the stack and jumps to this address. This is the way return works. Well, if you want to jump somewhere, you can push the return address yourself and then perform a return, which does the same thing as a jump.

In the crackme, something similar to this is done, but with a twist. The code pushes the address of the code we want to jump to (the beginning of the new decrypted code) but adding a constant value of 754841h to it. It then loads this value into eax, and subtracts the constant from it again. Finally, it copies the correct value onto the top of the stack and performs a return, telling the processor to pull this value off the stack and jump to it. Here's what it looks like in the disassembly:

0040121F	. 49	DEC ECX
00401220	. ^ 75 F8	JNZ SHORT Anti-cra.0040121A
00401222	. 68 A159B500	PUSH 0B559A1
00401227	. 8B0424	MOV EAX, DWORD PTR SS:[ESP]
0040122A	. 2D 41487500	SUB EAX, 754841
0040122F	. 8B0424	MOV DWORD PTR SS:[ESP], EAX
00401232	. C3	RET
00401233	. \$ 6A 00	PUSH 0

Granted, this is still a pretty easy version of this technique, but it gives you the idea for when you hit a tougher example.

Techniques Not in the Crackme

Here I will go over a couple of additional techniques that were not in the crackme, but that you should be familiar with.

Manually Loading Imports:

This technique is used quite often, especially in malware. It involves manually loading the imports that the binary needs, as opposed to having Windows load them for you. Imports are files like user32.dll and kernel32.dll, and the methods these files make available to our application. GetDlgItemText, MessageBox, and StrCmp are examples of these imported functions.

The reason for loading these manually is it removes the benefit that reversers can get by seeing the imports and where they are. For example, one of the first things we do, especially if there are no strings in the target, is to do a search for all intermodular calls. We then look for any suspicious calls that we know are usually associated with protections, and zero in on these. If the target loads imports manually, all you will see in this window are two processes, GetProcAddress and LoadLibrary. This is a sure-fire way of determining if this target loads imports manually.

*** There are additional techniques that can be used to even get rid of the GetProcAddress and LoadLibrary calls, making the intermodular calls window show no function calls. ***

The way an executable does this is it loads in the various DLLs itself, using LoadLibrary, then manually loads in the calls that it will use, saving pointers to these calls internally. When the target needs to call an

import, such as MessageBox (with or badboy message), instead of Olly showing something like this in the disassembly:

call user32.MessageBox

it will show something like this:

call [eax]

Obviously, the second method is much harder to follow, as we have no idea what function is being called. Also, having no starting points to follow that target's flow makes it much harder to reverse engineer.

The only way to combat this technique is to step the code from the beginning and try to figure out what each call is for. We can then manually label these calls in Olly, as a beginning to understanding, and ultimately cracking the target.

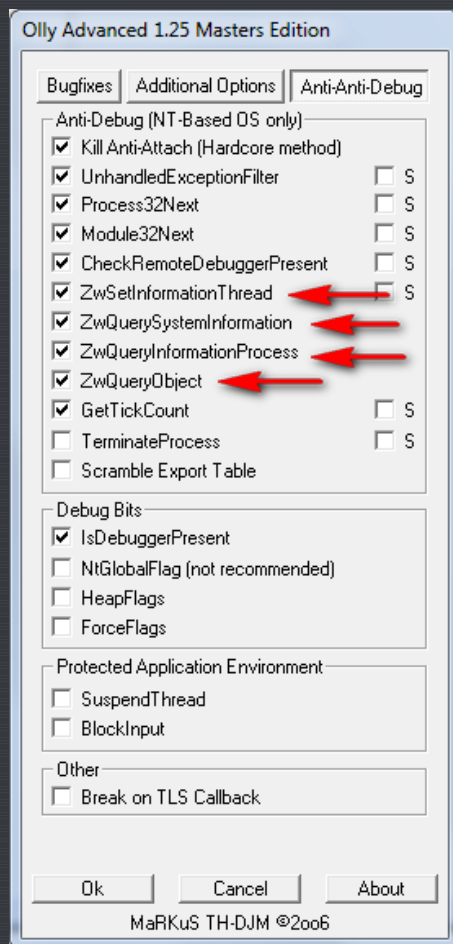
Alternatives to IsDebuggerPresent:

There are other ways of asking Windows if the current process is being debugged besides IsDebuggerPresent. One way is with the NtQuerySystemInformation or ZwQuerySystemInformation calls. The latter is an undocumented internal call that Windows uses directly, though both achieve the same result. The call definitions look like this:

ZwQuerySystemInformation(SystemKernelDebuggerInformation,

(PVOID) &DebuggerInfo, sizeof(DebuggerInfo), &ulReturnedLength);

Just keep in mind that if you run across either of these functions, it is attempting to figure out whether the app is being run in a debugger. Most anti-anti-debugging plugins will thwart this, as we see in OllyAdvanced:



The Trap Flag:

This technique is nice in that it detects any debugger, no matter how 'smart' it is. This involves setting the

trap flag in the current process and check whether an exception is raised. If an exception is not raised, you can assume that the debugger has 'swallowed' the exception to help us, and that the program is being single-stepped. Here is an example, using a combination of psuedo-code and assembly:

```
BOOL ExceptionFlagged = FALSE;
```

```
Try
```

```
{
```

```
pushfd
```

```
// Set the trap flag
```

```
or dword ptr [esp], 0x100
```

```
popfd
```

```
nop
```

```
}
```

```
exception handler:
```

```
{
```

```
ExceptionFlagged = TRUE;
```

```
}
```

```
if( ExceptionFlagged == FALSE )
```

```
print "A debugger has been found";
```

When reversing code that looks like this, the pushad and popad instructions should jump out at you as you don't normally see these instructions (unless we're dealing with packed code, but that comes later). Obviously, the fix to this is NOPping out the if(ExceptionFlagged == FALSE) check.

Checksums:

I had initially included a checksum routine in this crackme, as they are used heavily in the real world, but I wanted to minimize the complexity of the code. Checksums, or CRC Checks as they are more often called, are a method of checking for breakpoints and patches to the code. The basic idea is that, at some point in the application, we add up the byte values of all the opcodes in the binary and check this sum with a hard-coded value. This hard-coded value is obtained when the binary is completed, but before being published.

When we patch a binary, we obviously change one or more of these opcodes, so when we add in the opcodes for the patched instructions, the end value will not be the same. Therefore we can tell dynamically if the file has changed, and do what we wish once we know this (usually quit the app). This dissuades patches to the binary.

The way this technique detects breakpoints is in the way a debugger sets them. When you place a breakpoint on an instruction, the debugger replaces the opcode for this instruction with an interrupt opcode, namely 0xCC, and stores the original opcode internally. This allows the debugger to trap this interrupt, pausing the application and allowing us to step through the code. As we saw above, we can scan memory looking for this exact value of CCh (as we did in the breakpoint check routine in our crackme), but, as this instruction's opcode was changed by the debugger, our checksum will also not match.

Let's look at an example. Here is the disassembly of a simple call with opcodes:

```
75 34 JNZ 4010A0
```

Here, the 75h is the opcode for JNZ and the 34h is an offset from this instruction to jump to, meaning we will jump 34h bytes forward in the code from this instruction if the result is not zero. If we place a breakpoint on this JNZ instruction, even though Olly still shows the correct opcodes, in memory our instruction would really look like this:

```
CC 34 JNZ 4010A0
```

75h, the opcode for the JNZ instruction, has been replaced with CCh. This way, when the operating system hits a CCh opcode, it performs an interrupt, and Olly has been coded to trap this interrupt so that we regain control when the breakpoint is hit. When this happens, Olly copies the original value (75h) back into this address, replacing the CCh code with the real opcode.

Now, if we add the first two values in the above instructions we get $75h + 34h = A9h$, which is our hard-coded checksum code. If we ran the checksum again after the breakpoint is set, we see that we get the value $CCh + 34h = 100h$, which does not equal the value it's supposed to, namely A9. therefore, we now know that a breakpoint (or patch) has been set in this code, therefore the code is running in a debugger.

Conclusion

Anti-debugging is a very large field, and even though I've covered a number of them here, there are still far more out there, with new ones being developed all the time. If you wish to learn more, here is a list of some more in-depth coverage of these techniques, all available on the [tools](#) page of the LegendOfRandom site:

Anti-Debugging- A Developer's Perspective by Tyler Shields

Nice overview with descriptions of each type. Also talks about PEB and TEB types of techniques.

Anti Reverse Engineering Guide

A .chm file (Windows help file) with three contributors supplying three takes on anti-debug techniques. Two are by Josh Jackson and one by Nicolas Falliere. The one by Falliere (the last) is the most detailed.

General Method of Program Code Obfuscation by Gregory Wroblewski

A long, very detailed analysis of code obfuscation techniques. Includes a lot of the math behind obfuscation analysis along with test programs to test the various techniques.

OllyDbg Detection Tricks by Pumqara

A tutorial on some specific methods of detecting OllyDBG. Includes source files to investigate.

Ultimate Anti Debugging Reference by Peter Ferrie

Peter Ferrie basically wrote the book on anti-debugging techniques. He has been doing research for a long time, and has come out with many cutting-edge techniques. this is his opus devoted to anti-debugging. It contains every anti-debugging technique I've ever seen. If you're only going to have one resource, this is the one.

-Till next time

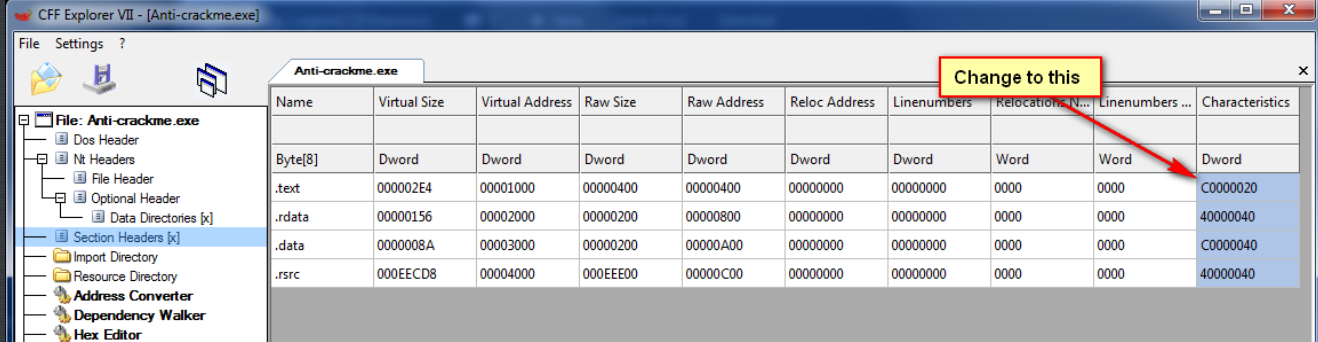
R4ndom

Addendum: Changing PE Section Attributes

One thing not mentioned earlier in the self-modifying code section was that in order to modify a binary's code dynamically, the characteristics of the .text section must be changed to allow writing to them.

The .text section (where the runnable code is generally kept) is almost always set as read-only. Since we are modifying this code, we must add the write attribute to this section. If this is not done, when a disassembler hits the instruction that changes data in the binary, you will get an exception and the program will crash.

In order to change the characteristics, I use CFF Explorer. First, compile your program to create the exe file (using RadASM, WinASM etc.) Then open the exe in CFF Explorer and click on the "Section Headers" tab in the directory tree:



When you load your version, the characteristics value will be set to 40000020. Double-click this field and change it to C0000040 as I've done. Select "File" -> "Save" and save the altered exe. Now when you run the application, the characteristics are set to "Execute as code", "Readable" and "Writable", allowing the binary to modify itself.