



## R4ndom's Tutorial #20A: Working With Visual Basic Binaries, Pt.

1

by R4ndom on Sep.02, 2012, under Beginner, Reverse Engineering, Tutorials

### Introduction

In this tutorial we will go over working with targets written in Visual Basic. Unfortunately, to become a well-rounded reverse engineer, we must know how to deal with these animals as there are many applications written in VB. Because this is a rather large subject, I will split it into two tutorials.

We will be looking at two crackmes, both included in the download of this tutorial. We will also be using VB Decompiler (the Lite version) which is included in the download.

As always, this tutorial, as well as all support files, can be downloaded on the [tutorials](#) page.

### Introducing Visual Basic

Visual Basic is an event-driven language. This means that instead of a program running from beginning to end, VB reacts to events that happen in a window. This is similar to Windows programming in that events take place and call methods that are registered to handle those events, but VB differs in that most of the processing and message creation is performed in a DLL file. This file is the Visual Basic 'runtime'.

The process of creating an application is a little different than, say, C++. You generally create a window (or dialog box) by dragging elements from a toolbox onto your window canvas. It is similar to C# .NET in this regard (and Delphi). Once you have your window built, you then create methods that will handle any events that can come from a user interacting with your windows contents; if a user clicks a button, the method you have made that handles the button event is called. If a user types in an edit box, the edit box method is called. Because the only code you are providing is the event code, most of the window's processing is done for you. All of this processing is done in a DLL file called "msvbvm60.dll", though the '60' may be different if using a different version of the runtime.

Another huge difference between Visual Basic and more traditional languages is a programmer has the option of compiling a VB application natively or in something called P-code. Native is simply assembly language, running natively on a processor, therefore OS and processor specific. P-code, on the other hand, is interpreted, much like Java and .NET, making it runnable on various operating systems. Interpreted means that, after compiling your VB application into P-code, when a user runs your application, something like a virtual machine is run, which interprets the P-code into native code for that specific operating system on-the-fly. When used, the p-code engine is a relatively simple machine that processes a series of "high-level" operation codes ("opcodes"). This engine is also stack-based, so very few arguments or functions are passed through registers.

The benefit of this is that if you install the VB runtime on, say, a Mac, then the P-code compiled application will be interpreted and run on a Mac. Switching to a Linux environment simply means running the Linux virtual machine (by installing the runtime), and voila, your app will run in Linux. Of course the downside is you take a speed hit as the code must be converted to native code before running.

Because VB applications can be compiled into P-code, the traditional debugging tools are a lot harder to use. Combine this with the fact that most of the time is spent in a DLL we don't care about, and it can be quite challenging. The good news is that there are a couple tools out there that will help us. We will be going over these shortly.

### Investigating The Target in Olly

When you first load a Visual Basic program in a debugger such as Olly, you will see a call is immediately

performed into the VB DLL, where it stays until another event happens. Because of this, VB programs are a little different to reverse engineer. The first thing you will notice is that the call stack is worthless; this is because most of the program's running time is within a DLL file, the VB runtime DLL. We don't care about this DLL, but we do care about the callback methods that handle events.

CPU - main thread, module AfKayAs\_

Address	Disassembly	Comment
00401170	\$ 68 D4674000	PUSH AfKayAs_004067D4
00401175	E8 F0FFFFFF	CALL <JMP.&MSUBVM50.#100>
0040117A	. 0000	ADD BYTE PTR DS:[EAX],AL
0040117C	. 0000	ADD BYTE PTR DS:[EAX],AL
0040117E	. 0000	ADD BYTE PTR DS:[EAX],AL
00401180	. 3000	XOR BYTE PTR DS:[EAX],AL
00401182	. 0000	ADD BYTE PTR DS:[EAX],AL
00401184	. 3000	CMF BYTE PTR DS:[EAX],AL
00401186	. 0000	ADD BYTE PTR DS:[EAX],AL
00401188	. 0000	ADD BYTE PTR DS:[EAX],AL
0040118A	. 0000	ADD BYTE PTR DS:[EAX],AL
0040118C	. 14 69	ADC AL,69
0040118E	. 65:BB DC42D311	MOV EBX,11D342DC
00401194	. BB	DB BB

Superfluous prefix

Beginning of VB program

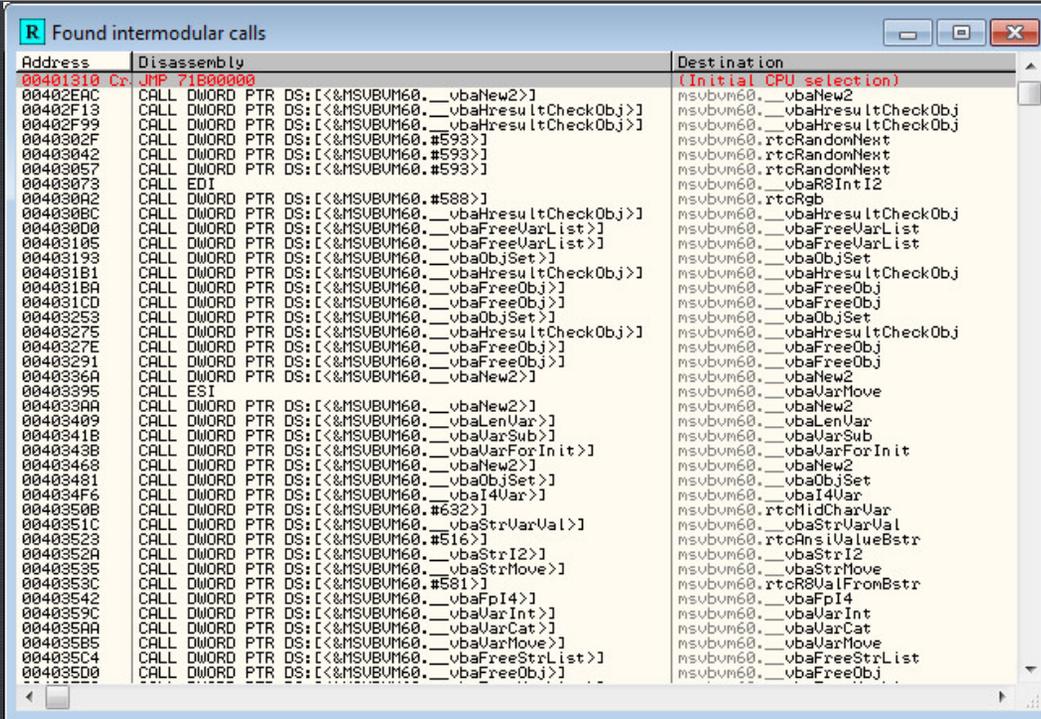
Another difference is in the way strings are handled. Because most of the message boxes, as well as all other window controls, are stored in resource sections, Olly won't display strings like a traditional C or C++ program. Therefore, using strings to find relevant sections of code is usually not an option:

Found strings are

Address	Disassembly	Text string
00401000	DD msvbvm60._vbaVarSub	(Initial CPU selection)
00401050	DD msvbvm60._adj_fdivr_m16i	ASCII "P=0"
00401058	DD msvbvm60._adj_fdivr_m16i	ASCII "P=0"
00401064	DD msvbvm60._vbaChkstk	ASCII "QWP="
004010A8	DD msvbvm60._adj_fdivr_m32i	ASCII "P=0"
004010AC	DD msvbvm60._adj_fdivr_m32i	ASCII "P=0"
00401125	ASCII "1g",0	
0040115D	ASCII "7g",0	
0040117D	ASCII "g",0	
00401346	ASCII "020430Progetto1",0	
00401356	ASCII "-C000-",0	
00401365	ASCII "p"5(?)",0	
004013C4	ASCII "Form1",0	
004013CE	ASCII "Leimcrackme",0	
004013E6	ASCII "lt",0	
0040144C	ASCII "LLL",0	
00401483	ASCII "fff",0	
00401493	ASCII "fff",0	
004014A5	ASCII "ff",0	
004014B2	ASCII "ffff",0	
004014D2	ASCII "ffff",0	
004014F2	ASCII "ffff",0	
00401513	ASCII "ff",0	
00401534	ASCII "ff",0	
00401553	ASCII "fff",0	
00401585	ASCII "f",0	
00401593	ASCII "fff",0	
004015B3	ASCII "fff",0	
004015F4	ASCII "f",0	
00401623	ASCII "fr=",0	
004016EF	ASCII "Form1",0	
004016F9	ASCII "S-",0	
00401717	ASCII "Text2",0	
00401733	ASCII "Text1",0	
0040174F	ASCII "Command2",0	
0040175C	ASCII "%Check",0	
00401779	ASCII "Command1",0	
00401786	ASCII "&About",0	

\*\*\*If you want to see the following data yourself in Olly, load CrackmeVB1.exe.\*\*\*

Another hindrance to reversing is the fact that the method calls are completely different in a VB executable. Instead of calls to such things as RegisterWindowEx and MessageBoxA, VB uses its own API calls, embedded in the runtime DLL:



Clicking through to one of these methods details the difference between VB and what we are used to:



As you can see, there are no helpful strings, no recognizable API calls.

Before we look at the tools at our disposal, let's see what the basic file structure of a VB executable is. I have loaded CrackmeVB1.exe, which is compiled in P-code. Scrolling to the top of the code in the disassembler view, we see the list of functions in the binary:

00401000	EA77A472	DD	msubvm60.__vbaUarSub
00401004	0705A272	DD	msubvm60.__vbaStrI2
00401008	8639A372	DD	msubvm60.__Cicos
0040100C	F909A372	DD	msubvm60.__adj_fptan
00401010	EE6AA472	DD	msubvm60.__vbaUarMove
00401014	3168A472	DD	msubvm60.__vbaFreeUar
00401018	9B6AA272	DD	msubvm60.__vbaLenBstr
0040101C	8DCCA172	DD	msubvm60.rtcRgb
00401020	6272A472	DD	msubvm60.__vbaFreeUarList
00401024	BA02A372	DD	msubvm60.__adj_fdiv_m64
00401028	C39FA172	DD	msubvm60.__vbaFreeObjList
0040102C	B770A272	DD	msubvm60.rtcAnsiUalaeBstr
00401030	4109A372	DD	msubvm60.__adj_fpreml
00401034	7492A172	DD	msubvm60.__vbaHresuLtCheckObj
00401038	AB6AA272	DD	msubvm60.__vbaLenUar
0040103C	6E02A372	DD	msubvm60.__adj_fdiv_m32
00401040	CC93A472	DD	msubvm60.__vbaUarForInit
00401044	05CDA172	DD	msubvm60.rtcRandomNext
00401048	32D1A172	DD	msubvm60.rtcMsgBox
0040104C	F19FA172	DD	msubvm60.__vbaObjSet
00401050	0603A372	DD	msubvm60.__adj_fdiv_m16i
00401054	08A0A172	DD	msubvm60.__vbaObjSetAddrRef
00401058	0604A372	DD	msubvm60.__adj_fdivr_m16i
0040105C	EE94A372	DD	msubvm60.__Cisin
00401060	2F70A272	DD	msubvm60.rtcMidCharUar
00401064	E6E2A372	DD	msubvm60.__vbaChkstk
00401068	7492A172	DD	msubvm60.EVENT_SINK_AddRef
0040106C	F697A472	DD	msubvm60.__vbaUarTstEq
00401070	F609A372	DD	msubvm60.__adj_fpatan
00401074	879BA072	DD	msubvm60.EVENT_SINK_Release
00401078	9395A372	DD	msubvm60.__Cisqrt
0040107C	859AA072	DD	msubvm60.EVENT_SINK_QueryInterface
00401080	6076A472	DD	msubvm60.__vbaUarMul
00401084	DF47A272	DD	msubvm60.__vbaExceptHandler
00401088	8906A372	DD	msubvm60.__adj_fpreml
0040108C	BA03A372	DD	msubvm60.__adj_fdivr_m64
00401090	1375A472	DD	msubvm60.__vbaFPEException
00401094	4813A272	DD	msubvm60.__vbaStrUarUal
00401098	7D63A272	DD	msubvm60.__vbaUarCat
0040109C	2B94A372	DD	msubvm60.__Cilog
004010A0	3722A172	DD	msubvm60.__vbaNew2

This is a reference for the runtime for the API calls that will be needed when the program is run.

Scrolling down a little we come to the jump table. This is similar to the jump table seen in most windows binaries and is there to help with code relocation:

00401190	00	DD	Crackme_00403CDF	
0040119C	DF3C4000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaChkstk>]	msubvm60.__vbaChkstk
004011A0	FF25 64104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaExceptHandler>]	msubvm60.__vbaExceptHandler; Struct
004011A4	FF25 90104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaFPEException>]	msubvm60.__vbaFPEException
004011B2	FF25 50104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdiv_m16i>]	msubvm60.__adj_fdiv_m16i
004011B8	FF25 3C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdiv_m32i>]	msubvm60.__adj_fdiv_m32i
004011BE	FF25 A8104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdiv_m32i_r>]	msubvm60.__adj_fdiv_m32i_r
004011C4	FF25 24104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdiv_m64i>]	msubvm60.__adj_fdiv_m64i
004011CA	FF25 B8104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdiv_r>]	msubvm60.__adj_fdiv_r
004011D0	FF25 58104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdivr_m16i>]	msubvm60.__adj_fdivr_m16i
004011D6	FF25 B4104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdivr_m32i>]	msubvm60.__adj_fdivr_m32i
004011DC	FF25 AC104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdivr_m32i_r>]	msubvm60.__adj_fdivr_m32i_r
004011E2	FF25 8C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fdivr_m64i>]	msubvm60.__adj_fdivr_m64i
004011E8	FF25 70104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fpatan>]	msubvm60.__adj_fpatan
004011EE	FF25 88104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fpreml>]	msubvm60.__adj_fpreml
004011F4	FF25 30104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fptan>]	msubvm60.__adj_fptan
004011FA	FF25 0C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__adj_fptan>]	msubvm60.__adj_fptan
00401200	FF25 04104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Ciatan>]	msubvm60.__Ciatan
00401206	FF25 08104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Cicos>]	msubvm60.__Cicos
0040120C	FF25 E8104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Ciexp>]	msubvm60.__Ciexp
00401212	FF25 9C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Cilog>]	msubvm60.__Cilog
00401218	FF25 5C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Cisin>]	msubvm60.__Cisin
0040121E	FF25 78104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Cisqrt>]	msubvm60.__Cisqrt
00401224	FF25 E0104000	JMP	DWORD PTR DS:[<&MSUBVM60.__Citan>]	msubvm60.__Citan
0040122A	FF25 DC104000	JMP	DWORD PTR DS:[<&MSUBVM60.__allmul>]	msubvm60.__allmul
00401230	FF25 EC104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaFreeObj>]	msubvm60.__vbaFreeObj
00401236	FF25 4C104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaObjSet>]	msubvm60.__vbaObjSet
0040123C	FF25 20104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaFreeUarList>]	msubvm60.__vbaFreeUarList
00401242	FF25 D0104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaR8IntI2>]	msubvm60.__vbaR8IntI2
00401248	FF25 1C104000	JMP	DWORD PTR DS:[<&MSUBVM60.#588>]	msubvm60.rtcRgb
0040124E	FF25 44104000	JMP	DWORD PTR DS:[<&MSUBVM60.#593>]	msubvm60.rtcRandomNext
00401254	FF25 34104000	JMP	DWORD PTR DS:[<&MSUBVM60.__vbaHresuLtCheckObj>]	msubvm60.__vbaHresuLtCheckObj

After this, we come to a vast sea of data. This is where the VB binary stores it's resources. Anything from strings, to buttons, to callbacks are stored in here. One thing to note is that Visual Basic uses the actual name of a callback; so if you want "MyButtonCallback" to handle the button event, that string will be used to reference it. Because of this, you will see the various callback names embedded in this resource section:

```

00401748 29 DB 29
00401749 00 DB 00
0040174A 00 DB 00
0040174B 00 DB 00
0040174C 03 DB 03
0040174D 08 DB 08
0040174E 00 DB 00
0040174F . 43 6F 6D 6D 61 6E ASCII "Command2",0
00401750 04 DB 04
00401751 01 DB 01
00401752 07 DB 07
00401753 00 DB 00
00401754 . 26 43 68 65 63 6E ASCII "%Check!",0
00401755 04 DB 04
00401756 00 DB 00
00401757 00 DB 00
00401758 00 DB 00
00401759 00 DB 00
0040175A 00 DB 00
0040175B 00 DB 00
0040175C . 43 6F 6D 6D 61 6E ASCII "Command1",0
0040175D 04 DB 04
0040175E 01 DB 01
0040175F 07 DB 07
00401760 00 DB 00
00401761 . 26 41 62 6F 75 74 ASCII "%About!",0
00401762 04 DB 04
00401763 00 DB 00
00401764 00 DB 00
00401765 00 DB 00
00401766 00 DB 00
00401767 00 DB 00
00401768 77 DB 77
00401769 01 DB 01
0040176A 11 DB 11
0040176B 02 DB 02
0040176C 00 DB 00
0040176D 00 DB 00
0040176E FF DB FF
0040176F 03 DB 03
00401770 28 DB 28
00401771 00 DB 00
00401772 00 DB 00
00401773 00 DB 00
00401774 02 DB 02
00401775 00 DB 00
00401776 00 DB 00
00401777 00 DB 00
00401778 . 43 6F 6D 6D 61 6E ASCII "Command",0
00401779 04 DB 04
0040177A 01 DB 01
0040177B 07 DB 07
0040177C 00 DB 00
0040177D . 26 41 62 6F 75 74 ASCII "%About",0
0040177E 04 DB 04
0040177F 00 DB 00
00401780 00 DB 00
00401781 00 DB 00
00401782 37 DB 37
00401783 05 DB 05
00401784 77 DB 77
00401785 01 DB 01
00401786 11 DB 11
00401787 01 DB 01
00401788 00 DB 00
00401789 00 DB 00
0040178A 00 DB 00
0040178B 00 DB 00
0040178C 00 DB 00
0040178D 00 DB 00
0040178E 00 DB 00
0040178F 00 DB 00
00401790 00 DB 00
00401791 00 DB 00
00401792 00 DB 00
00401793 00 DB 00
00401794 00 DB 00
00401795 00 DB 00
00401796 00 DB 00
00401797 00 DB 00
00401798 00 DB 00

```

The "Check It" button

The CheckIt callback name

The "About" button

The "About" callback

Scrolling down (much) further, we get to the actual event callbacks. These are the user generated callback methods to handle the various events. As you can see, there is no documentation as to which callback methods each is, though we will change this later with MAP files:

```

00402F3D 90 NOP
00402F3E 90 NOP
00402F3F 90 NOP
00402F40 > 55 PUSH EBP
00402F41 . 8BEC MOV EBP,ESP
00402F42 . 83EC 0C SUB ESP,0C
00402F43 . 68 A6114000 PUSH <JMP.&MSUBUM60,___vbaExceptHandler>
00402F44 . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00402F45 . 50 PUSH EAX
00402F46 . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
00402F47 . 83EC 0C SUB ESP,0C
00402F48 . 53 PUSH EBX
00402F49 . 56 PUSH ESI
00402F4A . 57 PUSH EDI
00402F4B . 8965 F4 MOV [LOCAL.3],ESP
00402F4C . C745 F8 00114000 MOV [LOCAL.2],Crackme_.00401100
00402F4D . 8B75 08 MOV ESI,[ARG.1]
00402F4E . 8BC6 MOV EAX,ESI
00402F4F . 83E0 01 AND EAX,1
00402F50 . 8945 FC MOV [LOCAL.1],EAX
00402F51 . 83E6 FE AND ESI,FFFFFFFE
00402F52 . 56 PUSH ESI
00402F53 . 8975 08 MOV [ARG.1],ESI
00402F54 . MOV ECX,DWORD PTR DS:[ESI]
00402F55 . CALL DWORD PTR DS:[ECX+4]
00402F56 . MOV EDX,DWORD PTR DS:[ESI]
00402F57 . 56 PUSH ESI
00402F58 . FF92 F8060000 CALL DWORD PTR DS:[EDX+6F8]
00402F59 . 85C0 TEST EAX,EAX
00402F5A . 7D 12 JGE SHORT Crackme_.00402F9F
00402F5B . 68 F8060000 PUSH 6F8
00402F5C . 68 C0254000 PUSH Crackme_.004025C0
00402F5D . 56 PUSH ESI
00402F5E . 50 PUSH EAX
00402F5F . FF15 34104000 CALL DWORD PTR DS:[<&MSUBUM60,___vbaHresuLtCheckObj>]
00402F60 . 7C 45 FC 00000000 MOV [LOCAL.1],0
00402F61 . 8B45 08 MOV EAX,[ARG.1]
00402F62 . 50 PUSH EAX
00402F63 . 8B08 MOV ECX,DWORD PTR DS:[EAX]
00402F64 . FF51 08 CALL DWORD PTR DS:[ECX+8]
00402F65 . 8B45 FC MOV EAX,[LOCAL.1]
00402F66 . 8B4D EC MOV ECX,[LOCAL.5]
00402F67 . 5F POP EDI
00402F68 . 5E POP ESI
00402F69 . 64:890D 00000000 MOV DWORD PTR FS:[0],ECX
00402F6A . 5B POP EBX
00402F6B . 8BE5 MOV ESP,EBP
00402F6C . 5D POP EBP
00402F6D . C2 0400 RETN 4
00402F6E 90 NOP
00402F6F 90 NOP
00402F70 90 NOP
00402F71 90 NOP
00402F72 90 NOP
00402F73 90 NOP
00402F74 90 NOP
00402F75 90 NOP
00402F76 90 NOP
00402F77 90 NOP
00402F78 90 NOP
00402F79 90 NOP
00402F7A 90 NOP
00402F7B 90 NOP
00402F7C 90 NOP
00402F7D 90 NOP
00402F7E 90 NOP
00402F7F 90 NOP
00402F80 90 NOP
00402F81 90 NOP
00402F82 90 NOP
00402F83 90 NOP
00402F84 90 NOP
00402F85 90 NOP
00402F86 90 NOP
00402F87 90 NOP
00402F88 90 NOP
00402F89 90 NOP
00402F8A 90 NOP
00402F8B 90 NOP
00402F8C 90 NOP
00402F8D 90 NOP
00402F8E 90 NOP
00402F8F 90 NOP
00402F90 90 NOP
00402F91 90 NOP
00402F92 90 NOP
00402F93 90 NOP
00402F94 90 NOP
00402F95 90 NOP
00402F96 90 NOP
00402F97 90 NOP
00402F98 90 NOP
00402F99 90 NOP
00402F9A 90 NOP
00402F9B 90 NOP
00402F9C 90 NOP
00402F9D 90 NOP
00402F9E 90 NOP
00402F9F 90 NOP
00402FA0 90 NOP
00402FA1 90 NOP
00402FA2 90 NOP
00402FA3 90 NOP
00402FA4 90 NOP
00402FA5 90 NOP
00402FA6 90 NOP
00402FA7 90 NOP
00402FA8 90 NOP
00402FA9 90 NOP
00402FAA 90 NOP
00402FAB 90 NOP
00402FAC 90 NOP
00402FAD 90 NOP
00402FAE 90 NOP
00402FAF 90 NOP
00402FB0 90 NOP
00402FB1 90 NOP
00402FB2 90 NOP
00402FB3 90 NOP
00402FB4 90 NOP
00402FB5 90 NOP
00402FB6 90 NOP
00402FB7 90 NOP
00402FB8 90 NOP
00402FB9 90 NOP
00402FBA 90 NOP
00402FBB 90 NOP
00402FBC 90 NOP
00402FBD 90 NOP
00402FBE 90 NOP
00402FBF 90 NOP
00402FC0 90 NOP
00402FC1 90 NOP
00402FC2 90 NOP
00402FC3 90 NOP
00402FC4 90 NOP
00402FC5 90 NOP
00402FC6 90 NOP
00402FC7 90 NOP
00402FC8 90 NOP
00402FC9 90 NOP
00402FCA 90 NOP
00402FCB 90 NOP
00402FCC 90 NOP
00402FCD 90 NOP
00402FCE 90 NOP
00402FCF 90 NOP
00402FD0 90 NOP
00402FD1 90 NOP
00402FD2 90 NOP
00402FD3 90 NOP
00402FD4 90 NOP
00402FD5 90 NOP
00402FD6 90 NOP
00402FD7 90 NOP
00402FD8 90 NOP
00402FD9 90 NOP
00402FDA 90 NOP
00402FDB 90 NOP
00402FDC 90 NOP
00402FDD 90 NOP
00402FDE 90 NOP
00402FDF 90 NOP
00402FE0 90 NOP
00402FE1 90 NOP
00402FE2 90 NOP
00402FE3 90 NOP
00402FE4 90 NOP
00402FE5 90 NOP
00402FE6 90 NOP
00402FE7 90 NOP
00402FE8 90 NOP
00402FE9 90 NOP

```

A callback method

Another callback method

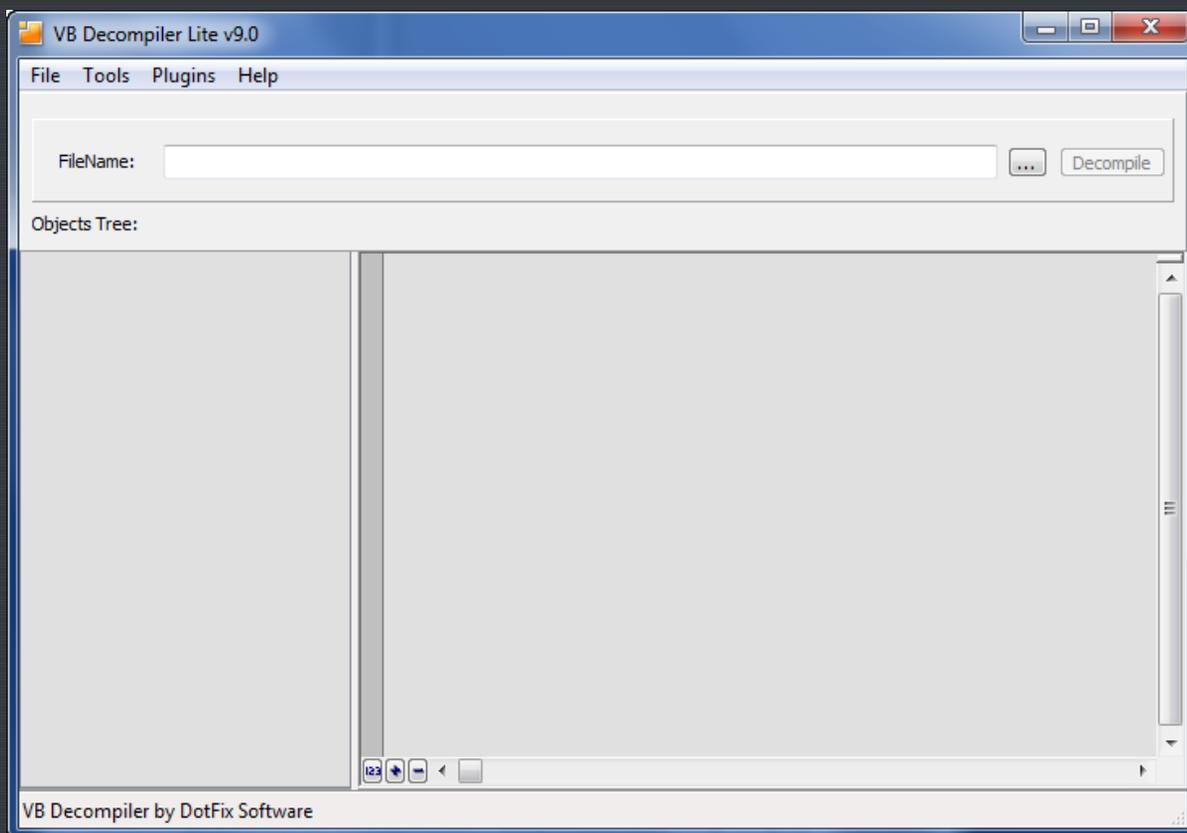
Lastly we come to the Import Address Table, or IAT. We will get \*much\* more familiar with this in the tutorials on unpacking:

00403D12	9E	DB 9E	
00403D13	9E	DB 9E	
00403D14	• 3C300000	DD 00003D3C	Struct 'IMAGE_IMPORT_DESCRIPTOR'
00403D18	• FFFFFFFF	DD FFFFFFFF	
00403D1C	• FFFFFFFF	DD FFFFFFFF	
00403D20	• 343E0000	DD 00003E34	
00403D24	• 00100000	DD 00001000	Struct 'IMAGE_IMPORT_DESCRIPTOR'
00403D28	• 00000000	DD 00000000	
00403D2C	• 00000000	DD 00000000	
00403D30	• 00000000	DD 00000000	
00403D34	• 00000000	DD 00000000	
00403D38	• 00000000	DD 00000000	
00403D3C	• 423E0000	DD 00003E42	Import lookup table for 'MSUBUM60.DLL'
00403D40	• 503E0000	DD 00003E50	
00403D44	• 5E3E0000	DD 00003E5E	
00403D48	• 683E0000	DD 00003E68	
00403D4C	• 763E0000	DD 00003E76	
00403D50	• 863E0000	DD 00003E86	
00403D54	• 963E0000	DD 00003E96	
00403D58	• 4C020080	DD 8000024C	
00403D5C	• A63E0000	DD 00003EA6	
00403D60	• BA3E0000	DD 00003EBA	
00403D64	• CA3E0000	DD 00003ECA	
00403D68	• 04020080	DD 80000204	
00403D6C	• DE3E0000	DD 00003EDE	
00403D70	• EC3E0000	DD 00003EEC	
00403D74	• 043F0000	DD 00003F04	
00403D78	• 123F0000	DD 00003F12	
00403D7C	• 223F0000	DD 00003F22	
00403D80	• 51020080	DD 80000251	
00403D84	• 53020080	DD 80000253	
00403D88	• 343F0000	DD 00003F34	
00403D8C	• 423F0000	DD 00003F42	
00403D90	• 543F0000	DD 00003F54	
00403D94	• 683F0000	DD 00003F68	
00403D98	• 703F0000	DD 00003F70	

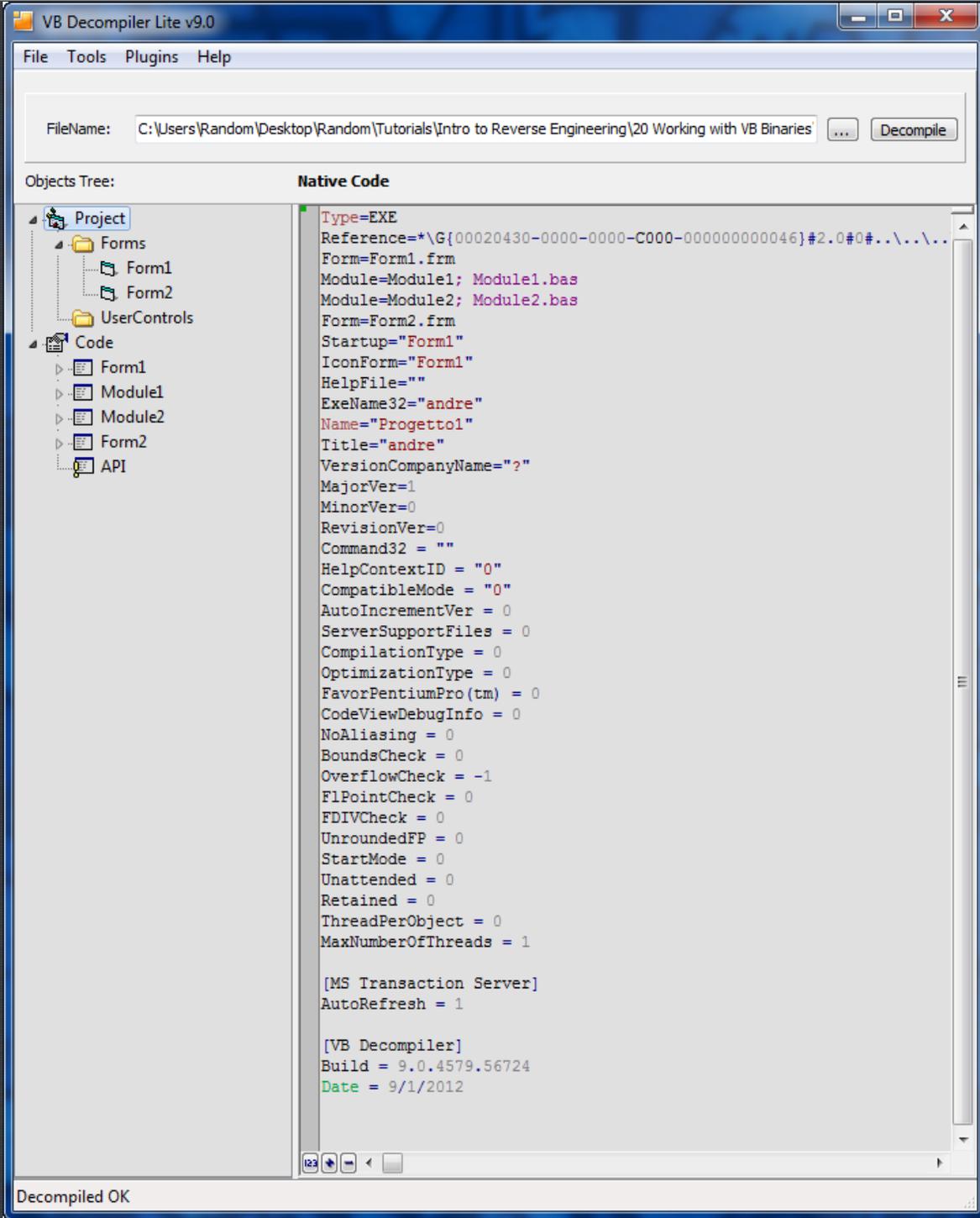
and that's pretty much it. Obviously, this isn't a heck of a lot of information to work with. Fortunately, we have a tool...

## VB Decompiler Lite

VB Decompiler is available in both a 'Lite' and 'Pro' version, the Lite version being free (and so, the one included with this tutorial). VB Decompiler allows us to decompile Visual Basic code, that has been converted into P-code, back into the original VB source code. Well, almost anyway 😊. It also allows us to view the resources embedded in the executable in a much friendlier format. Running VB Decompiler Lite, we first see the main screen:



Opening our first crackme, "CrackmeVB1.exe" and selecting the 'Decompile' button, we see the main project:



Most of this information is unimportant- mostly just file attributes etc. Notice, though, that in the project tree (under the 'Forms' folder) there are two forms, form1 and Form2. These are the resources associated with each form. Because there are two, we know that this application actually has two forms; One the main window and, in this case, one an about screen. Running that app confirms this:

## Form 1- Main Screen

## Form 2 - "About"

Leimcrackme.... Autore: nessuno  
(l'autore si vergogna di questo cosa  
e non vuole che il suo nome venga  
ulteriormente infangato.) Anzi dai ve  
lo dico.. il vero autore sono io,

OK

Leimcrackme

Whoever tries this out and never solves it means it's a big loser.... this isn't a big deal, so u must do that before doing anything else... if u don't manage to solve it... go and hide yourself for ever and ever! Amen.

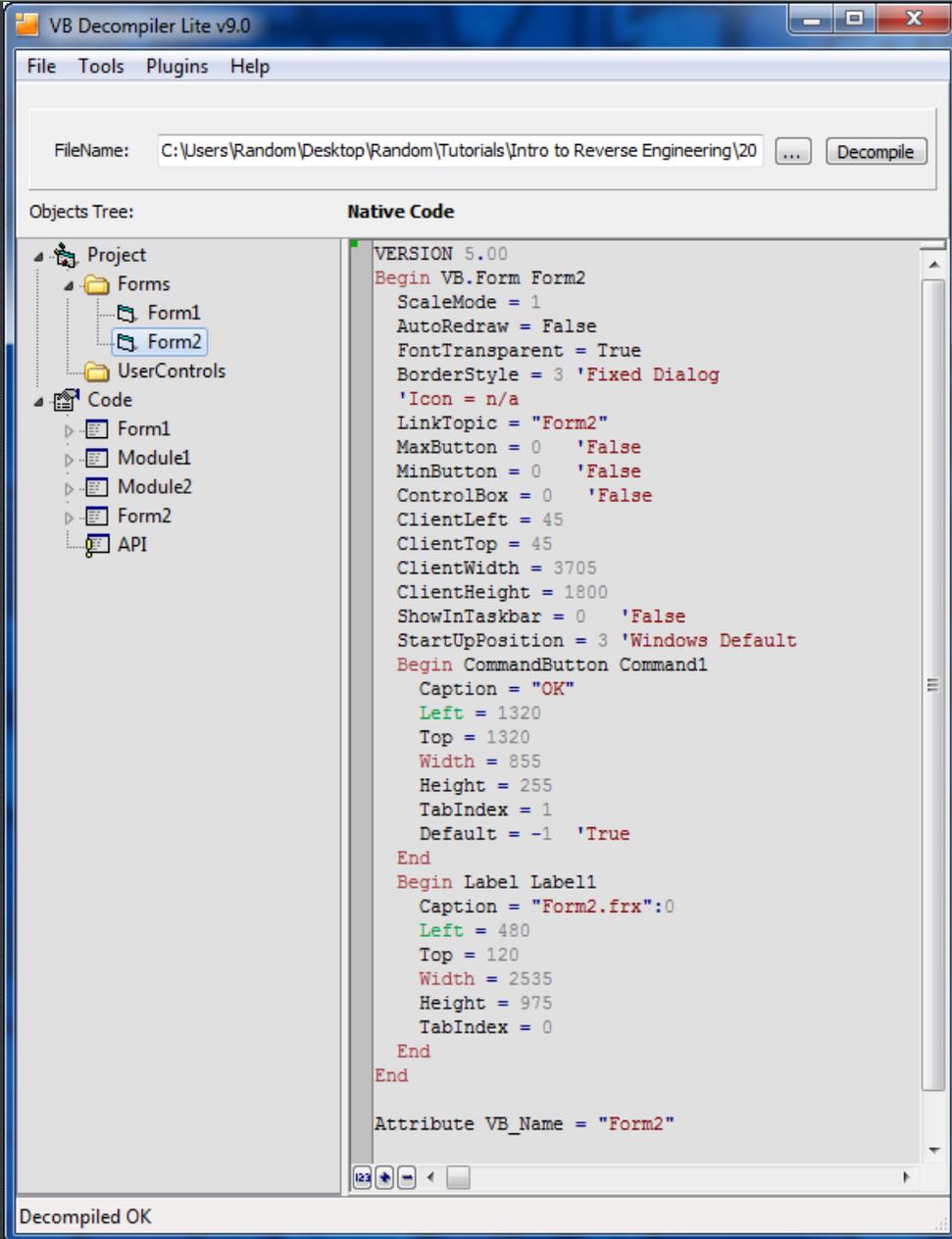
Name:

Serial:

Check! About

You will also notice two additional things when running this target; The about screen is in a different language, and you cannot click the "OK" button in the about screen. If any of you have followed my tutorials on modifying binaries, you will know that, of course, this is my favorite thing about this crackme 😊

If you double click on "Form2" in the "Forms" folder (in the project tree of VB Decompiler) we will see the various resources, along with attributes, for Form2:



Here we can see that there is one button with the text "OK", one label with the text in a different language, and one callback method for the "OK" button event called "Command1".

Double clicking on Form1 brings up the main window's attributes:

VB Decompiler Lite v0.4

File Tools Help

FileName: C:\Users\Random\Desktop\Random\Tutorials\Intro to Reverse Engineering\20 Working with \ Decompile

Objects Tree: Interface:

```

Project
├── Forms
│   ├── Form1
│   └── Form2
├── UserControls
├── Code
│   ├── Form1
│   ├── Module1
│   ├── Module2
│   ├── Form2
│   └── API
└── ...

```

```

ShowInTaskbar = 0 'false
StartupPosition = 3 'Windows Default
Begin VB.TextBox Text2
    Left = 1440
    Top = 1920
    Width = 2775
    Height = 285
    TabIndex = 4
End
Begin VB.TextBox Text1
    Left = 1440
    Top = 1440
    Width = 2775
    Height = 285
    TabIndex = 3
End
Begin VB.CommandButton Command2
    Caption = "&Check!"
    Left = 720
    Top = 2400
    Width = 1215
    Height = 375
    TabIndex = 2
End
Begin VB.CommandButton Command1
    Caption = "&About"
    Left = 2880
    Top = 2400
    Width = 1335
    Height = 375
    TabIndex = 1
End
Begin VB.Label Label2
    Caption = "Serial:"
    Index = 1
    Left = 360
    Top = 1920
    Width = 855
    Height = 255
    TabIndex = 6
    BackStyle = 0 'Transparent
End
Begin VB.Label Label2
    Caption = "Name:"
    Index = 0
    Left = 360
    Top = 1440
    Width = 855
    Height = 255
    TabIndex = 5
    BackStyle = 0 'Transparent
End
Begin VB.Label Label1
    Caption = "Whoever tries this out and never solves it means it's a big loser.... this isn't a big deal, so u must do that before doing anything else... if u don't manage to solve it... go and hide yourself for ever and ever! Amen."
    Left = 360
    Top = 360

```

Message Boxes

"Check!" button callback name

"Check!" button

"About" button

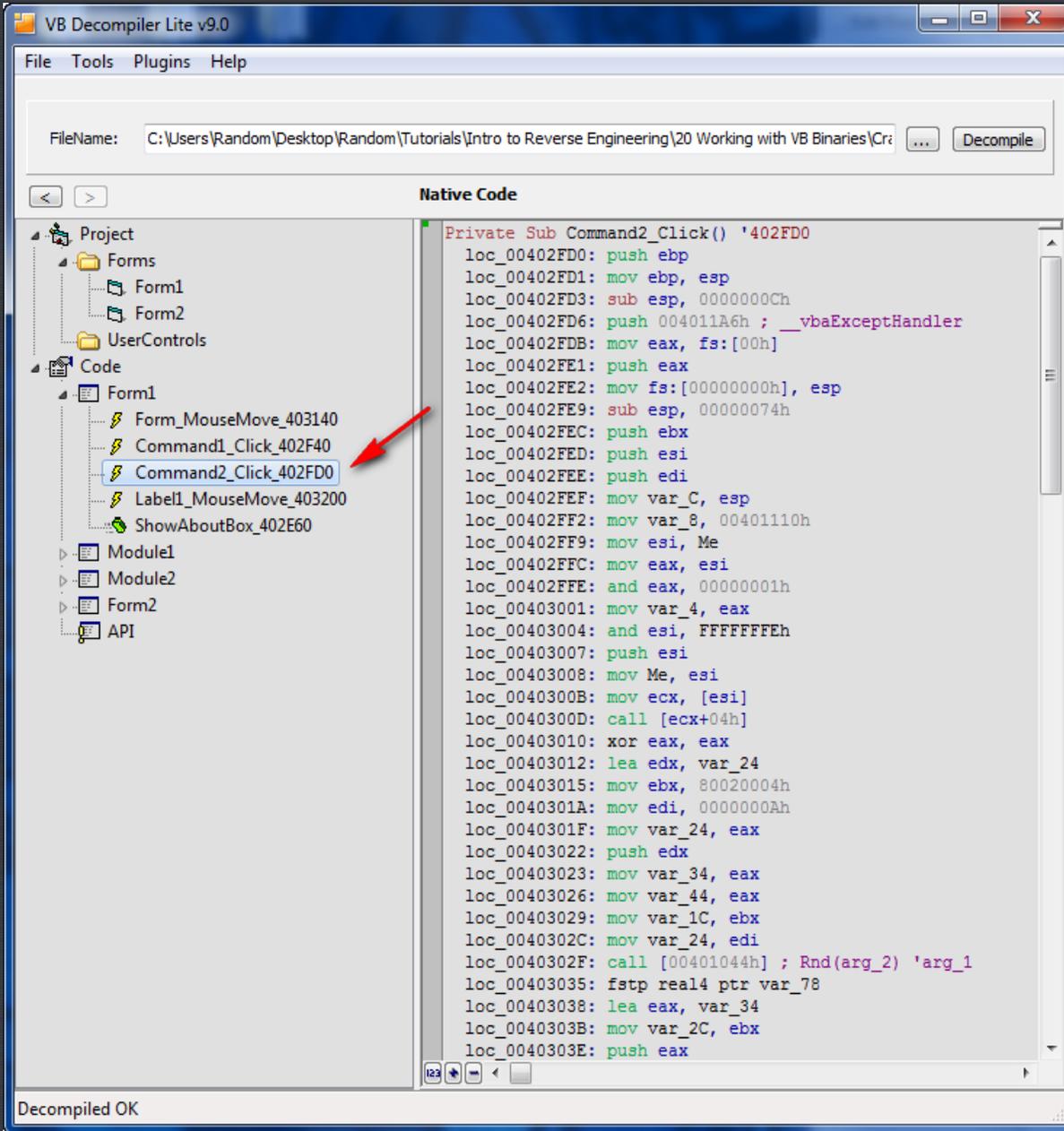
Text Boxes

Label

Decompiling OK

Now we know several important things about this crackme; the important button is called "Check!" and has a callback method with the name of "Command2", and Form1 is the main form we want to concern ourselves with. If you look down the tree, under the "Code" node, you will see the code that corresponds with the various forms. Opening the 'Form1' tree, we see that there are five callbacks, one for the "Checkit" button (Command2\_Click\_402FD0) and others for other buttons and mouse movements. If you run the target, you will see that the mouse movements callback is to change the color of the text when you hover over it.

What we want is Command2, as that's our callback:



Double clicking on this shows us the actual assembly code...

The important thing about this screen is the address of the callback. All we really wanted to use VB Decompiler for (in this case) is to find the address of the callback for the "Checkit" button, which we can see is 402FD0. Going to this address in Olly (with the target loaded) shows us the beginning of the callback function:

CPU - main thread, module CrackmeV

00402FCE	90	NOP	
00402FCF	90	NOP	
00402FD0	55	PUSH EBP	Beginning of callback
00402FD1	8BEC	MOV EBP,ESP	
00402FD3	83EC 0C	SUB ESP,0C	
00402FD6	68 A6114000	PUSH <JMP.&MSUBUM60...vbaExceptionHandler>	SE handler installation
00402FDB	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
00402FE1	50	PUSH EAX	CrackmeU.0040240A
00402FE2	64:8925 00000000	MOV DWORD PTR FS:[0],ESP	
00402FE3	83EC 74	SUB ESP,74	
00402FEC	53	PUSH EBX	
00402FED	56	PUSH ESI	
00402FEE	57	PUSH EDI	
00402FEF	8965 F4	MOV DWORD PTR SS:[EBP-C],ESP	
00402FF2	C745 F8 10114000	MOV DWORD PTR SS:[EBP-8],CrackmeU.00401110	
00402FF9	8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]	
00402FFC	8BC6	MOV EAX,ESI	
00402FFE	83E0 01	AND EAX,1	
00403001	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	CrackmeU.0040240A
00403004	83E6 FE	AND ESI,FFFFFFFE	
00403007	56	PUSH ESI	
00403008	8975 08	MOV DWORD PTR SS:[EBP+8],ESI	
0040300B	8B0E	MOV ECX,DWORD PTR DS:[ESI]	
0040300D	FF51 04	CALL DWORD PTR DS:[ECX+4]	
00403010	33C0	XOR EAX,EAX	CrackmeU.0040240A
00403012	8D55 DC	LEA EDX,DWORD PTR SS:[EBP-24]	
00403015	BB 04002004	MOV EBX,80020004	
0040301A	BF 0A000000	MOV EDI,0A	
0040301F	8945 DC	MOV DWORD PTR SS:[EBP-24],EAX	CrackmeU.0040240A
00403022	52	PUSH EDX	MSUBUM60.72953E28
00403023	8945 CC	MOV DWORD PTR SS:[EBP-34],EAX	CrackmeU.0040240A
00403026	8945 BC	MOV DWORD PTR SS:[EBP-44],EAX	CrackmeU.0040240A
00403029	895D E4	MOV DWORD PTR SS:[EBP-1C],EBX	CrackmeU.0040240A
0040302C	897D DC	MOV DWORD PTR SS:[EBP-24],EDI	

If you set a BP here, run the target, and enter a username and serial, you will see that, after clicking the "Checkit" button, Olly pauses at our callback. We have now found our main registration callback code!!!

## VB Decompiler Pro

I wanted to show what the actual P-code looks like, and for that we need VB Decompiler Pro. Unfortunately, this application requires that you buy it (... 😞 ...) to use this function. Looking at the same code in VB Decompiler Pro looks like this:

```

Private Sub Command2_Click() '402FD0
    loc_00402FF2: var_8 = &H401110
    loc_00403029: var_1C = 80020004h
    loc_0040302C: var_24 = 10
    loc_00403035: var_78 = Rnd()
    loc_0040303B: var_2C = 80020004h
    loc_0040303F: var_34 = 10
    loc_00403048: var_7C = Rnd()
    loc_0040304E: var_3C = 80020004h
    loc_00403054: var_44 = 10
    loc_004030A2: Var_Ret_1=RGB(CInt(CInt(CInt(@CInt(%StkVar1))))),Me,esi)
    loc_004030AA: Me.BackColor = Var_Ret_1
    loc_004030D9: Proc_00403800(var_34, var_44, var_24)
    loc_004030E1: If Proc_00403800(var_34, var_44, var_24) <> 0 Then GoTo loc_004030E8
    loc_004030E3: Proc_004032C0(Proc_00403800(var_34, var_44, var_24), edx, ecx)
    loc_004030E8: var_4 = 0
    loc_004030F5: GoTo loc_0040310F
    loc_0040310E: Exit Sub
    loc_0040310F: Exit Sub
End Sub
  
```

Here, we can see the actual P-code method for the callback. First, several variables are set up. The background is changed at 4030AA, a procedure is called at 4030D9 (and it looks pretty interesting), and then what is probably our magic compare/jump is performed at address 4030E1. We can see that if the results of calling the procedure at 403800 are true, we will then jump to 4030E8. If not, we will fall through and perform the instructions beginning at address 4030E2. Taking a little time, we could actually find the patch this way, though I personally like going back to Olly to do it, as it doesn't hurt my brain so much.



changing the zero flag for the jump at address 4030E1. Unfortunately, this doesn't display anything. This means we want to take a closer look at the call to address 4032C0 at address 4030E3. Placing a BP here, restarting the target, and stepping in, we see the main decryption routine:

```

004032C0 $ 55      PUSH EBP
004032C1 . 8BEC    MOV EBP,ESP
004032C3 . 83EC 08 SUB ESP,8
004032C6 . 68 A6114000 PUSH <JMP.&MSUBUM60.__vbaExceptionHandler> SE handler installation
004032CB . 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
004032D1 . 50      PUSH EAX
004032D2 . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
004032D9 . 81EC 58010000 SUB ESP,158
004032DF . 53      PUSH EBX
004032E0 . 57      PUSH ESI
004032E1 . 57      PUSH EDI
004032E2 . 8965 F8 MOV DWORD PTR SS:[EBP-8],ESP
004032E5 . C745 FC 40114000 MOV DWORD PTR SS:[EBP-4],CrackmeU.00401140
004032EC . A1 10504000 MOV EAX,DWORD PTR DS:[405010]
004032F1 . 33FF    XOR EDI,EDI
004032F3 . 3BC7    CMP EAX,EDI
004032F5 . 897D E0 MOV DWORD PTR SS:[EBP-20],EDI
004032F8 . 897D D0 MOV DWORD PTR SS:[EBP-30],EDI
004032FB . 897D C0 MOV DWORD PTR SS:[EBP-40],EDI
004032FE . 897D B0 MOV DWORD PTR SS:[EBP-50],EDI
00403301 . 897D AC MOV DWORD PTR SS:[EBP-54],EDI
00403304 . 897D A8 MOV DWORD PTR SS:[EBP-58],EDI
00403307 . 897D A4 MOV DWORD PTR SS:[EBP-5C],EDI
0040330A . 897D 94 MOV DWORD PTR SS:[EBP-6C],EDI
0040330D . 897D 84 MOV DWORD PTR SS:[EBP-7C],EDI
00403310 . 898D 74FFFFFF MOV DWORD PTR SS:[EBP-8C],EDI
00403316 . 898D 64FFFFFF MOV DWORD PTR SS:[EBP-9C],EDI
0040331C . 898D 54FFFFFF MOV DWORD PTR SS:[EBP-AC],EDI
00403322 . 898D 44FFFFFF MOV DWORD PTR SS:[EBP-BC],EDI
00403328 . 898D 34FFFFFF MOV DWORD PTR SS:[EBP-CC],EDI
0040332E . 898D 24FFFFFF MOV DWORD PTR SS:[EBP-DC],EDI
00403334 . 898D 14FFFFFF MOV DWORD PTR SS:[EBP-EC],EDI
00403339 . 898D 04FFFFFF MOV DWORD PTR SS:[EBP-FC],EDI
00403346 . 898D F4FFFFFF MOV DWORD PTR SS:[EBP-10C],EDI
0040334C . 898D E4FFFFFF MOV DWORD PTR SS:[EBP-11C],EDI
0040334C . 898D C4FFFFFF MOV DWORD PTR SS:[EBP-13C],EDI
00403352 . 898D B4FFFFFF MOV DWORD PTR SS:[EBP-14C],EDI
00403358 . 898D A4FFFFFF MOV DWORD PTR SS:[EBP-15C],EDI
0040335E . 75 15   JNZ SHORT CrackmeU.00403375
00403360 . 68 10504000 PUSH CrackmeU.00405010
00403365 . 68 481F4000 PUSH CrackmeU.00401F48
0040336A . FF15 A0104000 CALL DWORD PTR DS:[<&MSUBUM60.__vbaNew2>] MSUBUM60.__vbaNew2
00403370 . A1 10504000 MOV EAX,DWORD PTR DS:[405010]
00403375 . 8B08    MOV ECX,DWORD PTR DS:[EAX]
00403377 . 50      PUSH EAX
00403378 . 8B91 00000000 CALL DWORD PTR DS:[ECX+900]
0040337E . 8B35 10104000 MOV ESI,DWORD PTR DS:[<&MSUBUM60.__vbaVarMove] MSUBUM60.__vbaVarMove
00403384 . BB 09000000 MOV EBX,9
00403389 . 8D55 94 LEA EDX,DWORD PTR SS:[EBP-6C]
0040338C . 8D4D B0 LEA ECX,DWORD PTR SS:[EBP-50]
0040338F . 8945 9C MOV DWORD PTR SS:[EBP-64],EAX
00403392 . 895D 94 MOV DWORD PTR SS:[EBP-6C],EBX
00403395 . FFD6    CALL ESI
00403397 . A1 10504000 MOV EAX,DWORD PTR DS:[405010]
0040339C . 3BC7    CMP EAX,EDI
0040339E . 75 15   JNZ SHORT CrackmeU.004033B5
004033A0 . 68 10504000 PUSH CrackmeU.00405010
004033A5 . 68 481F4000 PUSH CrackmeU.00401F48

```

As we will see shortly, there are some very standard method calls in VB that should be memorized. Scrolling down the code, we see one of these at address 403644:

```

00403628 . E8 E0000000 CALL CrackmeU.00403710
0040362D . E8 DE000000 CALL CrackmeU.00403710
00403632 . E8 D9000000 CALL CrackmeU.00403710
00403637 . E8 D4000000 CALL CrackmeU.00403710
0040363C . 8D4D C0 LEA ECX,DWORD PTR SS:[EBP-40]
0040363F . 8D55 D0 LEA EDX,DWORD PTR SS:[EBP-30]
00403642 . 51      PUSH ECX
00403643 . 52      PUSH EDX
00403644 . FF15 6C104000 CALL DWORD PTR DS:[<&MSUBUM60.__vbaVarTstEq] MSUBUM60.__vbaVarTstEq
0040364A . 66:85C0 TEST AX,AX
0040364D . 74 0D   JE SHORT CrackmeU.0040365C
0040364F . E8 CC000000 CALL CrackmeU.00403720
00403654 . 9B      WAIT
00403655 . 68 FE364000 PUSH CrackmeU.004036FE
0040365A . EB 6E   JMP SHORT CrackmeU.004036CA
0040365C . E8 BF030000 CALL CrackmeU.00403A20
00403661 . 9B      WAIT
00403662 . 68 FE364000 PUSH CrackmeU.004036FE
00403667 . EB 61   JMP SHORT CrackmeU.004036CA
00403669 . 8D45 A8 LEA EAX,DWORD PTR SS:[EBP-58]
0040366C . 8D4D AC LEA ECX,DWORD PTR SS:[EBP-54]
0040366F . 50      PUSH EAX
00403670 . 51      PUSH ECX
00403671 . 6A 02   PUSH 2
00403673 . FF15 B0104000 CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeStrList] MSUBUM60.__vbaFreeStrList
00403679 . 83C4 0C ADD ESP,0C
0040367C . 8D4D A4 LEA ECX,DWORD PTR SS:[EBP-5C]
0040367F . FF15 EC104000 CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeObj] MSUBUM60.__vbaFreeObj
00403685 . 8D95 14FFFFFF LEA EDX,DWORD PTR SS:[EBP-EC]
00403688 . 8D85 24FFFFFF LEA EAX,DWORD PTR SS:[EBP-DC]
00403691 . 52      PUSH EDX
00403692 . 8D8D 34FFFFFF LEA ECX,DWORD PTR SS:[EBP-CC]

```

vbaVarTstEq is like StrCmp in native code- it checks two entities to see if they match. Highlighting the call down three lines at address 40364F and clicking "Enter", Olly follows the call and we see we're on the right track:

0040371F	90	NOP	
00403720	55	PUSH EBP	
00403721	8BEC	MOV EBP,ESP	
00403723	83EC 08	SUB ESP,8	
00403726	68 A6114000	PUSH <JMP.&MSUBUM60,___vbaExceptionHandler>	SE handler installation
0040372B	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
00403731	50	PUSH EAX	
00403732	64:8925 00000000	MOV DWORD PTR FS:[0],ESP	
00403739	81EC 84000000	SUB ESP,84	
0040373F	53	PUSH EBX	MSUBUM60.___vbaVarAdd
00403740	56	PUSH ESI	MSUBUM60.___vbaVarMul
00403741	57	PUSH EDI	
00403742	8965 F8	MOV DWORD PTR SS:[EBP-8],ESP	
00403745	0745 FC 50114000	MOV DWORD PTR SS:[EBP-4],CrackmeU.00401150	
0040374C	B9 04000280	MOV ECX,20020004	
00403751	B8 0A000000	MOV EAX,0A	
00403756	894D B8	MOV DWORD PTR SS:[EBP-48],ECX	
00403759	894D C8	MOV DWORD PTR SS:[EBP-38],ECX	
0040375C	894D D8	MOV DWORD PTR SS:[EBP-28],ECX	
0040375F	8D55 A0	LEA EDX,DWORD PTR SS:[EBP-60]	
00403762	8D4D E0	LEA ECX,DWORD PTR SS:[EBP-20]	
00403765	C745 E0 00000000	MOV DWORD PTR SS:[EBP-20],0	
0040376C	8945 B0	MOV DWORD PTR SS:[EBP-50],EAX	
0040376F	8945 C0	MOV DWORD PTR SS:[EBP-40],EAX	
00403772	8945 D0	MOV DWORD PTR SS:[EBP-30],EAX	
00403775	C745 A8 00284000	MOV DWORD PTR SS:[EBP-58],CrackmeU.004028D0	UNICODE "You did it. I won't congratulate with you, i
0040377C	C745 A0 00000000	MOV DWORD PTR SS:[EBP-60],0	
00403783	FF15 C8104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarDup]	MSUBUM60.___vbaVarDup
00403789	8D45 B0	LEA EAX,DWORD PTR SS:[EBP-50]	
0040378C	8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
0040378F	50	PUSH EAX	
00403790	8D55 B0	LEA EDX,DWORD PTR SS:[EBP-30]	

So we know we must make the code execute to address 403644. Looking above this at the various jumps, we find the following JE at address 40344F:

00403435	50	PUSH EAX	
00403436	8D55 E0	LEA EDX,DWORD PTR SS:[EBP-20]	
00403439	51	PUSH ECX	
0040343A	52	PUSH EDX	
0040343B	FF15 40104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarForInit]	MSUBUM60.___vbaVarForInit
00403441	8B3D 80104000	MOV EDI,DWORD PTR DS:[&MSUBUM60.___vbaVarMul]	MSUBUM60.___vbaVarMul
00403447	8B1D C4104000	MOV EBX,DWORD PTR DS:[&MSUBUM60.___vbaVarAdd]	MSUBUM60.___vbaVarAdd
0040344D	85C0	TEST EAX,EAX	
0040344F	0F84 C9010000	JE CrackmeU.0040361E	Jumps to goodboy
00403455	A1 10504000	MOV EAX,DWORD PTR DS:[405010]	
0040345A	85C0	TEST EAX,EAX	
0040345C	75 15	JNZ SHORT CrackmeU.00403473	
0040345E	68 10504000	PUSH CrackmeU.00405010	
00403463	68 481F4000	PUSH CrackmeU.00401F48	
00403468	FF15 A0104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaNew2]	MSUBUM60.___vbaNew2

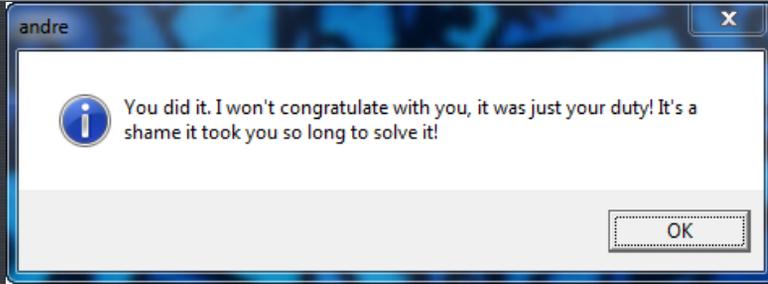
which jumps to the area of code we want:

0040361E	83C4 18	ADD ESP,18	
0040361F	8D3D A4FEFFFF	LEA ECX,DWORD PTR SS:[EBP-15C]	
0040361E	8D95 B4FEFFFF	LEA EDX,DWORD PTR SS:[EBP-14C]	
0040361E	8D45 E0	LEA EAX,DWORD PTR SS:[EBP-20]	
00403610	51	PUSH ECX	
00403611	52	PUSH EDX	
00403612	50	PUSH EAX	
00403613	FF15 E4104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarForNext]	MSUBUM60.___vbaVarForNext
00403619	E9 2FFFFFFFFF	JMP CrackmeU.0040344D	Jumps to here
0040361E	E8 E0000000	CALL CrackmeU.00403710	
00403623	E8 E0000000	CALL CrackmeU.00403710	
00403628	E8 E3000000	CALL CrackmeU.00403710	
0040362D	E8 DE000000	CALL CrackmeU.00403710	
00403632	E8 D9000000	CALL CrackmeU.00403710	
00403637	E8 D4000000	CALL CrackmeU.00403710	
0040363C	8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
0040363F	8D55 D0	LEA EDX,DWORD PTR SS:[EBP-30]	
00403642	51	PUSH ECX	
00403643	52	PUSH EDX	
00403644	FF15 6C104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarTstEq]	MSUBUM60.___vbaVarTstEq
0040364A	66:85C0	TEST AX,AX	
0040364D	74 0D	JE SHORT CrackmeU.0040365C	Shows goodboy
0040364F	E8 CC000000	CALL CrackmeU.00403720	
00403654	9B	WAIT	
00403655	68 FE364000	PUSH CrackmeU.004036FE	
0040365A	EB 6E	JMP SHORT CrackmeU.004036CA	
0040365C	E8 BF030000	CALL CrackmeU.00403A20	
00403661	9B	WAIT	

So let's place a BP at address 40344F, run the target, and change the zero flag to force the jump:

00403610	51	PUSH ECX	
00403611	52	PUSH EDX	
00403612	50	PUSH EAX	
00403613	FF15 E4104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarForNext]	MSUBUM60.___vbaVarForNext
00403619	E9 2FFFFFFFFF	JMP CrackmeU.0040344D	
0040361E	E8 E0000000	CALL CrackmeU.00403710	We land here after forcing the jump
00403623	E8 E0000000	CALL CrackmeU.00403710	
00403628	E8 E3000000	CALL CrackmeU.00403710	
0040362D	E8 DE000000	CALL CrackmeU.00403710	
00403632	E8 D9000000	CALL CrackmeU.00403710	
00403637	E8 D4000000	CALL CrackmeU.00403710	
0040363C	8D4D C0	LEA ECX,DWORD PTR SS:[EBP-40]	
0040363F	8D55 D0	LEA EDX,DWORD PTR SS:[EBP-30]	
00403642	51	PUSH ECX	
00403643	52	PUSH EDX	
00403644	FF15 6C104000	CALL DWORD PTR DS:[&MSUBUM60.___vbaVarTstEq]	MSUBUM60.___vbaVarTstEq
0040364A	66:85C0	TEST AX,AX	
0040364D	74 0D	JE SHORT CrackmeU.0040365C	
0040364F	E8 CC000000	CALL CrackmeU.00403720	
00403654	9B	WAIT	
00403655	68 FE364000	PUSH CrackmeU.004036FE	
0040365A	EB 6E	JMP SHORT CrackmeU.004036CA	

Now, stepping down to the JE instruction at 40364D, we obviously want to stop this from jumping over our call to the goodboy. Changing the zero flag when we land here, we see that we have in fact cracked the target:



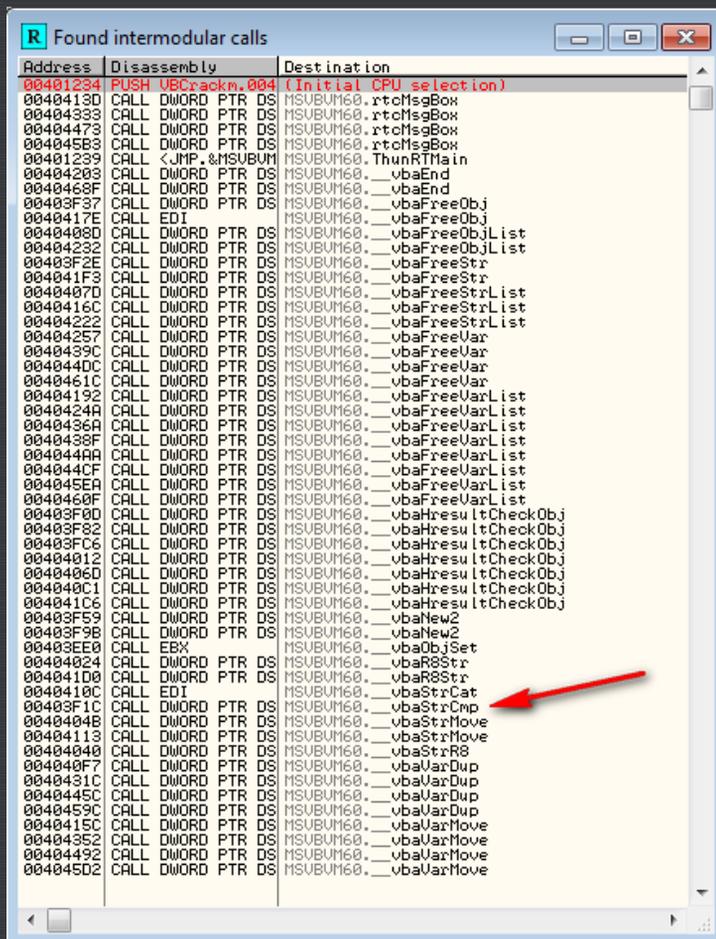
## Frequently Called Methods

As stated earlier, there are some methods that are called a lot when looking at protection schemes:

`_vbaVarTstEq`  
`_vbaVarTstNe`  
`_vbaVarCmpEq`  
`_vbaStrCmp`  
`_vbaStrComp`  
`_vbaStrCompVar`

9 out of 10 times, one of these routines will be used to compare a serial with the correct one. One of these, `_vbaVarTstEq`, was used in the previous crackme.

Go ahead and load CrackmeVB2.exe into Olly. Performing a search of intermodular calls, we see one of our suspicious calls:



Here we see the call to `_vbaStrCmp`. Looking up the `String.Compare` method call in the Visual Basic API, we see that it takes two strings as arguments and returns an int. The return value is either -1, 0 (for equals) and 1, depending on if the first is greater than or less than the second, or zero if they are equal. This is what the call looks like in VB:

```

'Declaration
Public Shared Function Compare ( _
    strA As String, _
    strB As String _
) As Integer
'Usage
Dim strA As String
Dim strB As String
Dim returnValue As Integer

returnValue = String.Compare(strA, strB)

```

Double clicking this call in Olly, we jump to where this call is performed.

00403EF9	7D 18	JE SHORT UBCrackn.00403F18	
00403EFB	88D 48FFFFFF	MOV ECX, DWORD PTR SS:[EBP-B8]	
00403F01	68 A0000000	PUSH 0A0	
00403F06	68 F8374000	PUSH UBCrackn.004037F8	
00403F08	51	PUSH ECX	
00403F0C	50	PUSH EAX	
00403F0D	FF15 28104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaHresultCheckObj	kernel32.BaseThreadInitThunk MSUBUM60.__vbaHresultCheckObj
00403F13	> 8B55 08	MOV EDX, DWORD PTR SS:[EBP-28]	
00403F16	52	PUSH EDX	
00403F17	68 0C384000	PUSH UBCrackn.0040380C	UBCrackn.<ModuleEntryPoint> UNICODE "g7*2+'&1,3"
00403F1C	FF15 4C104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaStrCmp>]	MSUBUM60.__vbaStrCmp
00403F22	8BF8	MOV EDI, EAX	
00403F24	8D4D 08	LEA ECX, DWORD PTR SS:[EBP-28]	kernel32.BaseThreadInitThunk
00403F27	F7DF	NEG EDI	
00403F29	1BFF	SBB EDI, EDI	
00403F2B	47	INC EDI	
00403F2C	F7DF	NEG EDI	
00403F2E	FF15 B0104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeStr>]	MSUBUM60.__vbaFreeStr
00403F34	8D4D 00	LEA ECX, DWORD PTR SS:[EBP-30]	
00403F37	FF15 B4104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeObj>]	MSUBUM60.__vbaFreeObj

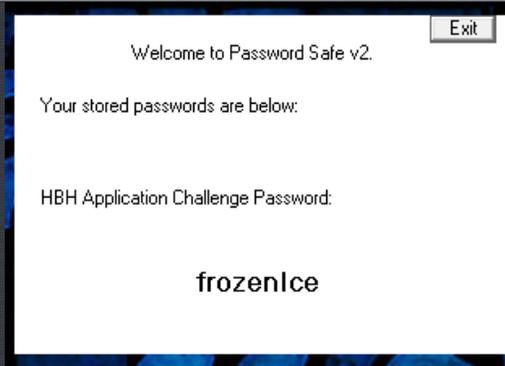
Let's set a BP on this line and run the target:



Entering a password (I entered '12121212') and clicking OK, Olly breaks right where we want him to:

00403F06	68 F8374000	PUSH UBCrackn.004037F8	
00403F08	51	PUSH ECX	ntdll.77866500
00403F0C	50	PUSH EAX	
00403F0D	FF15 28104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaHresultCheckObj	MSUBUM60.__vbaHresultCheckObj
00403F13	> 8B55 08	MOV EDX, DWORD PTR SS:[EBP-28]	
00403F16	52	PUSH EDX	
00403F17	68 0C384000	PUSH UBCrackn.0040380C	UNICODE "g7*2+'&1,3"
00403F1C	FF15 4C104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaStrCmp>]	MSUBUM60.__vbaStrCmp
00403F22	8BF8	MOV EDI, EAX	
00403F24	8D4D 08	LEA ECX, DWORD PTR SS:[EBP-28]	
00403F27	F7DF	NEG EDI	
00403F29	1BFF	SBB EDI, EDI	
00403F2B	47	INC EDI	
00403F2C	F7DF	NEG EDI	
00403F2E	FF15 B0104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeStr>]	MSUBUM60.__vbaFreeStr
00403F34	8D4D 00	LEA ECX, DWORD PTR SS:[EBP-30]	
00403F37	FF15 B4104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaFreeObj>]	MSUBUM60.__vbaFreeObj
00403F3D	66:85FF	TEST DI, DI	
00403F40	> 0F84 8B000000	JE UBCrackn.00403FD1	
00403F46	A1 24504000	MOV EAX, DWORD PTR DS:[405024]	
00403F4B	85C0	TEST EAX, EAX	
00403F4D	> 75 10	JNZ SHORT UBCrackn.00403F5F	
00403F4F	68 24504000	PUSH UBCrackn.00405024	
00403F54	68 A82C4000	PUSH UBCrackn.00402CA8	
00403F59	FF15 78104000	CALL DWORD PTR DS:[<&MSUBUM60.__vbaNew2>]	MSUBUM60.__vbaNew2
00403F5F	> 8B35 24504000	MOV ESI, DWORD PTR DS:[405024]	
00403F65	6A FF	PUSH -1	
00403F67	56	PUSH ESI	
00403F68	8B06	MOV EAX, DWORD PTR DS:[ESI]	UBCrackn.00405A64
00403F6A	FF90 BC010000	CALL DWORD PTR DS:[EAX+1BC]	

Looking down a little bit at address 403f40, we see our wonderful compare/jump instruction. Stepping down to there and changing the zero flag, then running the target, we see that this was our simplest crack yet 😊 :



There is a lot to take in here, but the most important thing is to mess around some on your own and discover how this stuff works on your own. I have included a crackme that we will be going over in the next tutorial (`crackmeVB3.exe`), so that you may try your hand at it. Following the same steps in this tutorial will solve this crackme as well.

In the next tutorial we will go over Smartcheck and the Point-H method, as well as creating MAP files.

-Till next time

R4ndom