

R4ndom's Tutorial #18: Time Trials and Memory Breakpoints

by R4ndom on Aug.22, 2012, under Beginner, Reverse Engineering, Tutorials

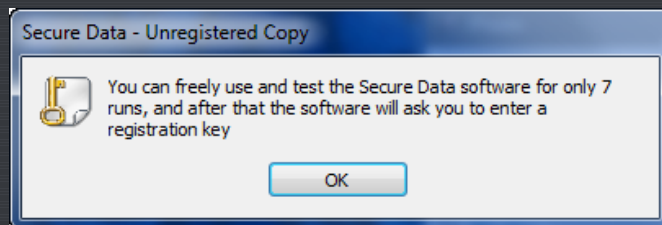
Introduction

Time trials are limitations on an application where you only get a certain amount of days or tries before the app stops working. Usually, an application will give you 30 days to try it, after which, it will be disabled. Sometimes, while cracking an application, it is worthwhile to reverse engineer the time trial code as it's easier to find the registration process when the app is still in trial mode. Also, if you're lazy, you can simply patch the time trial code and nothing else, giving yourself unlimited time to 'try' the app.

For this tutorial, in order to protect author's and their work, I have downloaded the worst reviewed "file hider" on CNET. Out of 1300 programs of this type, this program was rated dead last. It has been downloaded twice. I believe that the author has stopped any support of it. The name of the app is appropriately named "Secure Data – Hide a File into an Image.exe". I have shortened the name to "SecureData.exe" just to save myself the typing. I have also not included all of the DLLs for this app, so it will not work completely, but will be fine for this tutorial (How's that, teddy Rodgers?).

Loading the app

Running the app we immediately see the time trial:



Yours will probably have a different number of runs available to you as I couldn't remember how many was default with this app 😊 Clicking OK and we come to the main screen:



Clicking the "About" guy with no face we see:



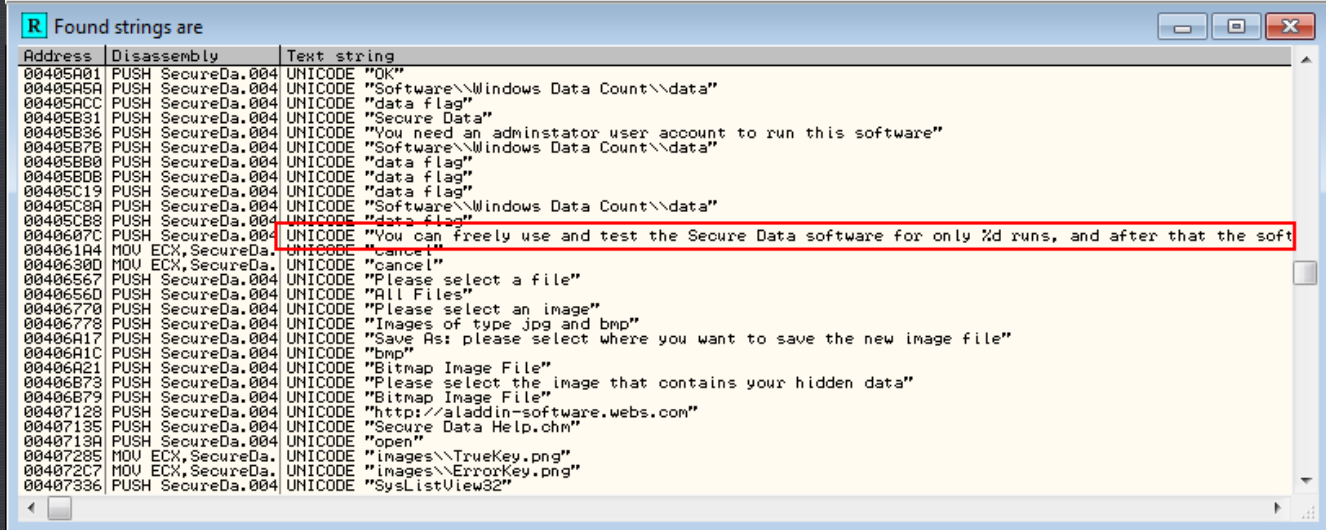
and typing in a key we get the message:



We've seen enough. Let's load the program in Olly:

CPU - main thread, module SecureDa				
004086A4	E8 304A0000	CALL SecureDa.004080D9		
004086A9	E9 16FEFFFF	JMP SecureDa.004084C4		
004086AE	55	PUSH EBP		
004086AF	8BEC	MOV EBP, ESP		
004086B1	81EC 28030000	SUB ESP, 328		
004086B7	A3 C8D64100	MOV DWORD PTR DS:[41D6C8], EAX	kernel32.BaseThreadInitThunk	
004086BC	890D C4D64100	MOV DWORD PTR DS:[41D6C4], ECX		
004086C2	8915 C0D64100	MOV DWORD PTR DS:[41D6C0], EDI	SecureDa.<ModuleEntryPoint>	
004086C8	891D BCD64100	MOV DWORD PTR DS:[41D6BC], EBX		
004086CE	8935 B8D64100	MOV DWORD PTR DS:[41D6B8], ESI		
004086D4	893D B4D64100	MOV DWORD PTR DS:[41D6B4], EDI		
004086DA	66:8C15 E0D64100	MOV WORD PTR DS:[41D6E0], SS		
004086E1	66:8C0D D4D64100	MOV WORD PTR DS:[41D6D4], CS		
004086E8	66:8C1D B0D64100	MOV WORD PTR DS:[41D6B0], DS		
004086EF	66:8C05 ACD64100	MOV WORD PTR DS:[41D6AC], FS		
004086F6	66:8C25 A8D64100	MOV WORD PTR DS:[41D6A8], FS		
004086FD	66:8C2D A4D64100	MOV WORD PTR DS:[41D6A4], GS		
00408704	9C	PUSHFD		
00408705	8F05 D8D64100	POP DWORD PTR DS:[41D6D8]	kernel32.77E5ED6C	
00408708	8B45 00	MOV EAX, DWORD PTR SS:[EBP]		
0040870E	A3 CCD64100	MOV DWORD PTR DS:[41D6CC], EAX	kernel32.BaseThreadInitThunk	
00408713	8B45 04	MOV EAX, DWORD PTR SS:[EBP+4]	ntdll.77D3377B	
00408716	A3 D0D64100	MOV DWORD PTR DS:[41D6D0], EAX	kernel32.BaseThreadInitThunk	
0040871B	8D45 00	LEA EAX, [ARG.1]		
0040871E	A3 DCD64100	MOV DWORD PTR DS:[41D6DC], EAX	kernel32.BaseThreadInitThunk	
00408723	8B85 E0FCFFFF	MOV EAX, [LOCAL.200]		
00408729	C705 18D64100 0	MOV DWORD PTR DS:[41D618], 1000		
00408733	A1 D0D64100	MOV EAX, DWORD PTR DS:[41D6D0]		
00408738	A3 CCD64100	MOV DWORD PTR DS:[41D6CC], EAX	kernel32.BaseThreadInitThunk	
0040873D	C705 C0D64100 0	MOV DWORD PTR DS:[41D5C0], C000		
00408747	C705 C4D64100 0	MOV DWORD PTR DS:[41D5C4], 1		

Searching for strings, we find the message for the time trial nag screen:



Though, we don't see any strings for the registration screen.

***One thing that is important to keep in mind is that every string in an app that will be used is not always in memory all the time. If the author of a program has put any effort into protecting the app, most important strings will not be decrypted until they need to be used. This is obviously the case here; when the registration screen needs the strings to tell the user that the key was wrong, those strings will be decrypted at that time. Later in the tutorial, we will see that this list of strings will change dramatically, and a whole new set of strings will be present in memory. ***

The Time Trial

The important thing for a reverse engineer to know about time trials is that the app MUST remember the amount of tries/days left after quitting and restarting the program. That means that certain data has to be stored persistently somewhere. The most obvious candidates for storing this data are the registry and a file on the hard drive.

Most of the time, this data is not altered, as it is a common misconception among application writers that someone will not want to go through the trouble of finding it (I guess...). Unfortunately for them, it is usually very easy to find this data. The easiest way is to look at the strings and search for either a registration path or a file path. These stick out like a sore thumb in the strings window. A registry path looks like this:

Software\\AppName\\Key

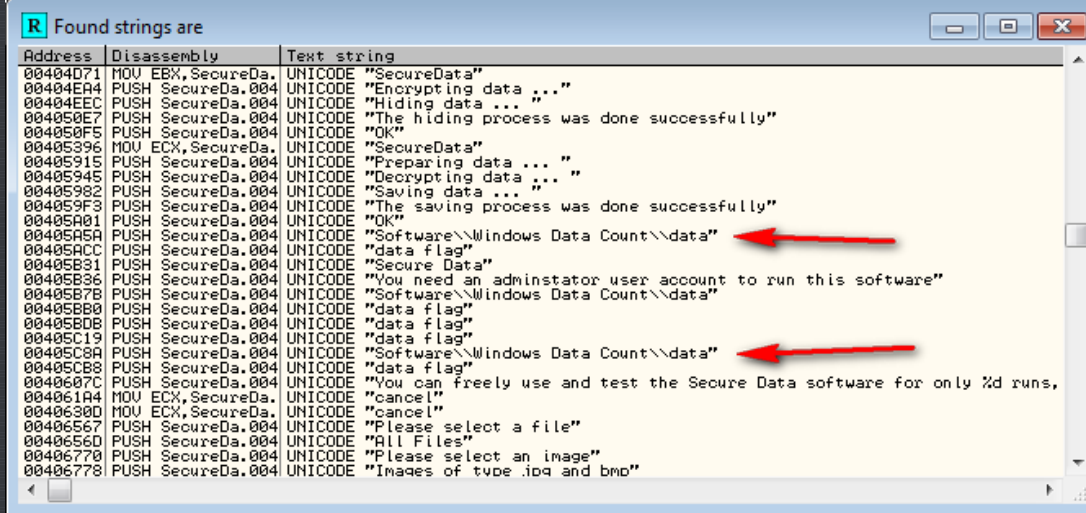
while a file path looks like this:

AppName\\DataFileName.ini or AppName\\DataFileName.dat

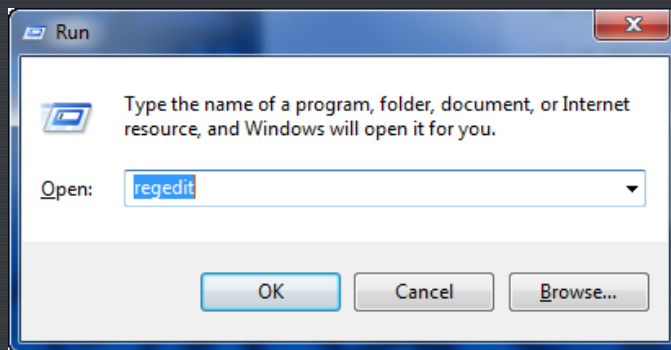
and many times will have something like %WINDOWS% in the name around it, pointing to the Windows install directory.

You can also use Search for all intermodular calls, as long as the app is not too large. You will either notice the CreateFileExA type method calls, or the RegSetValueExA type calls, depending on whether the app stores the data on disk or in the registry. Unfortunately, this app has both as it creates files on the drive to hide other files.

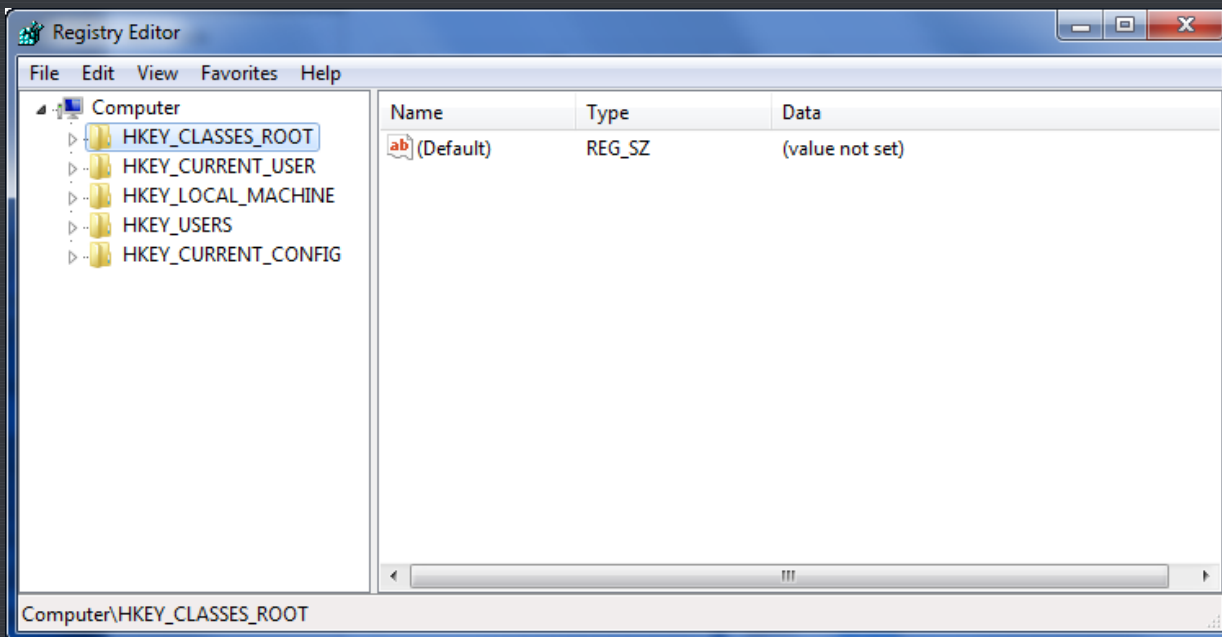
Looking in the strings window again we find a reference to a registry key:



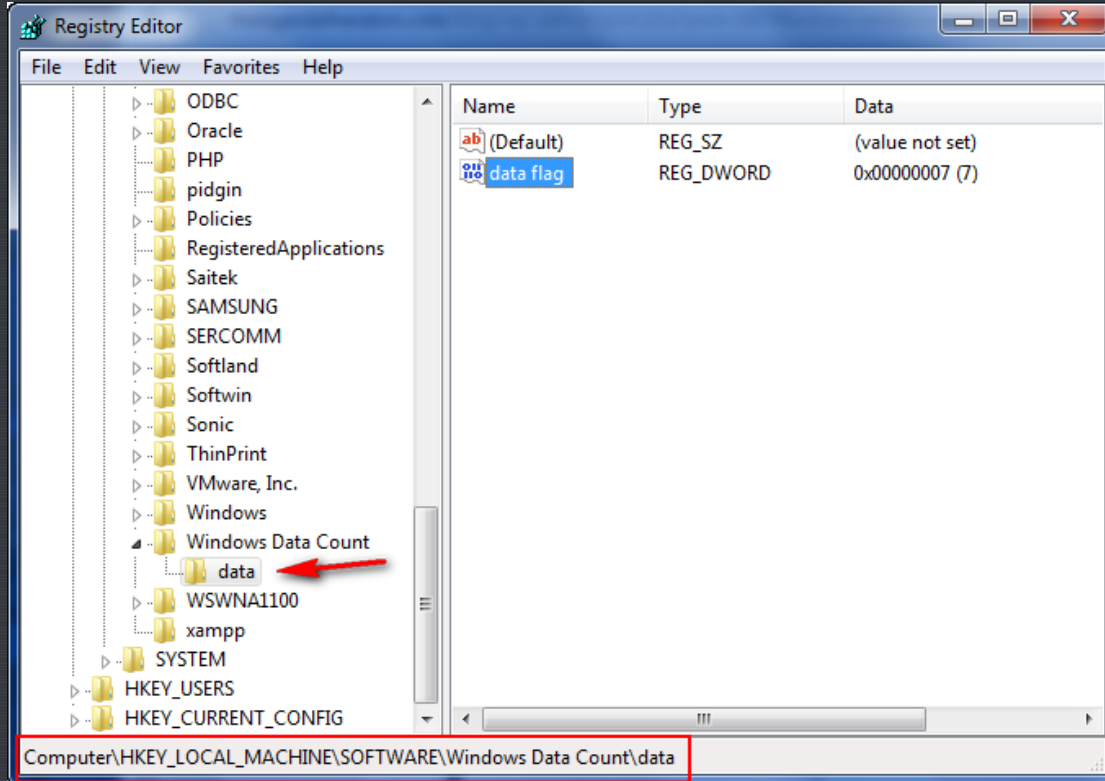
I will not go into a full detailed explanation of the registry. The info you need to know is that the registry is a normal file-tree structured database and you access it by using regedit (built into Windows). To run regedit, simply open a run prompt (WindowsKey-R) and type in "regedit":



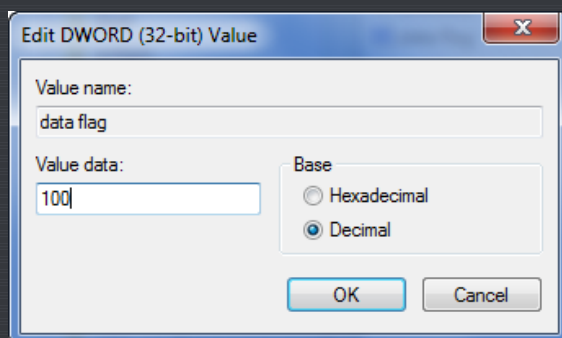
Here, we can see the top 5 folders of the registry:

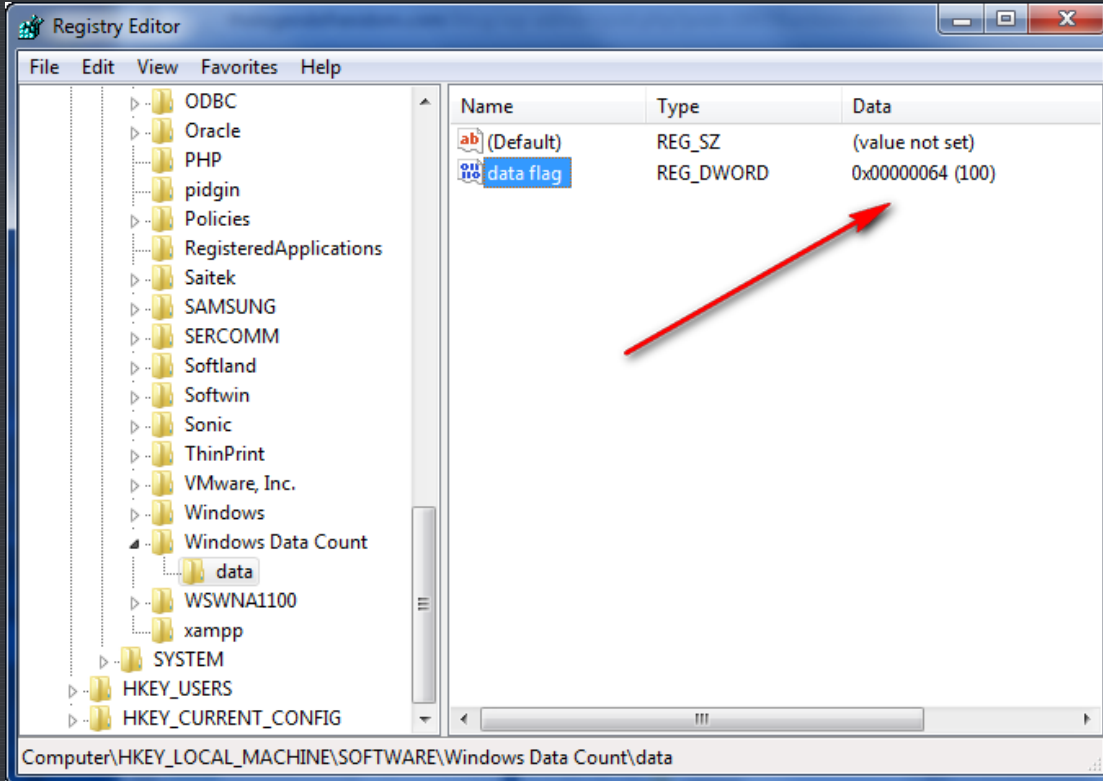


Our string does not contain the top root key in it, so opening each one of these folders, opening the "Software" key and looking for the "Windows Data Count" folder reveals it in the HKEY_LOCAL_MACHINE folder:

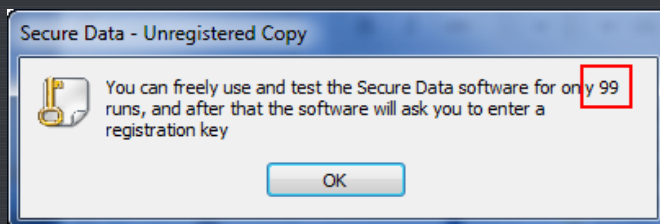


We can see on the left that the "data flag" key has a value of 7 (in my case- in yours it will be different). This looks very suspicious. Let's change it and see what happens. Right-click on it and choose "Modify". Enter a value of 100, making sure the decimal flag is on, and save it:





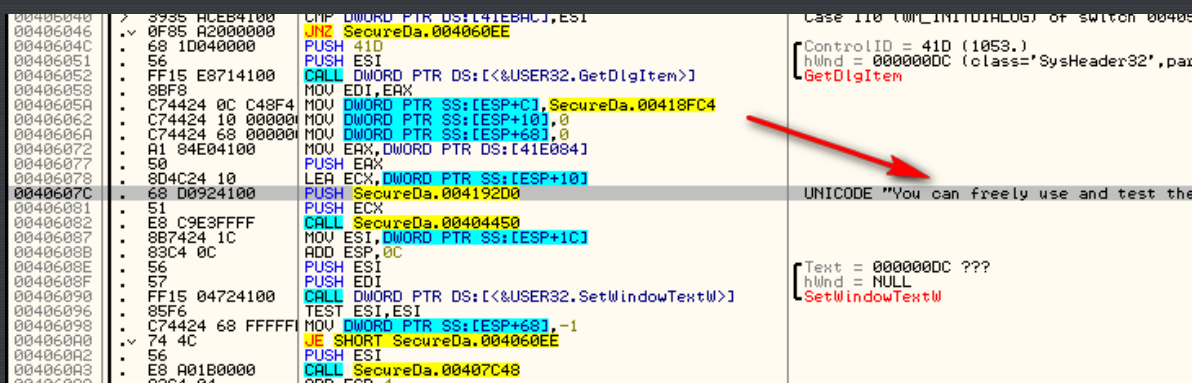
Now run the app:



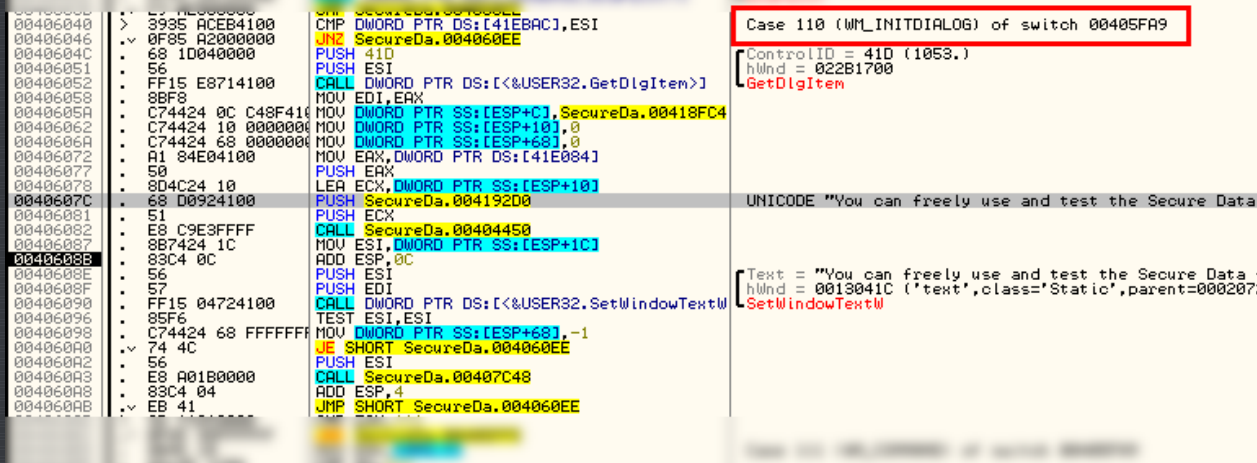
Well, that's a little easier to deal with 😊.

Investigating the Binary

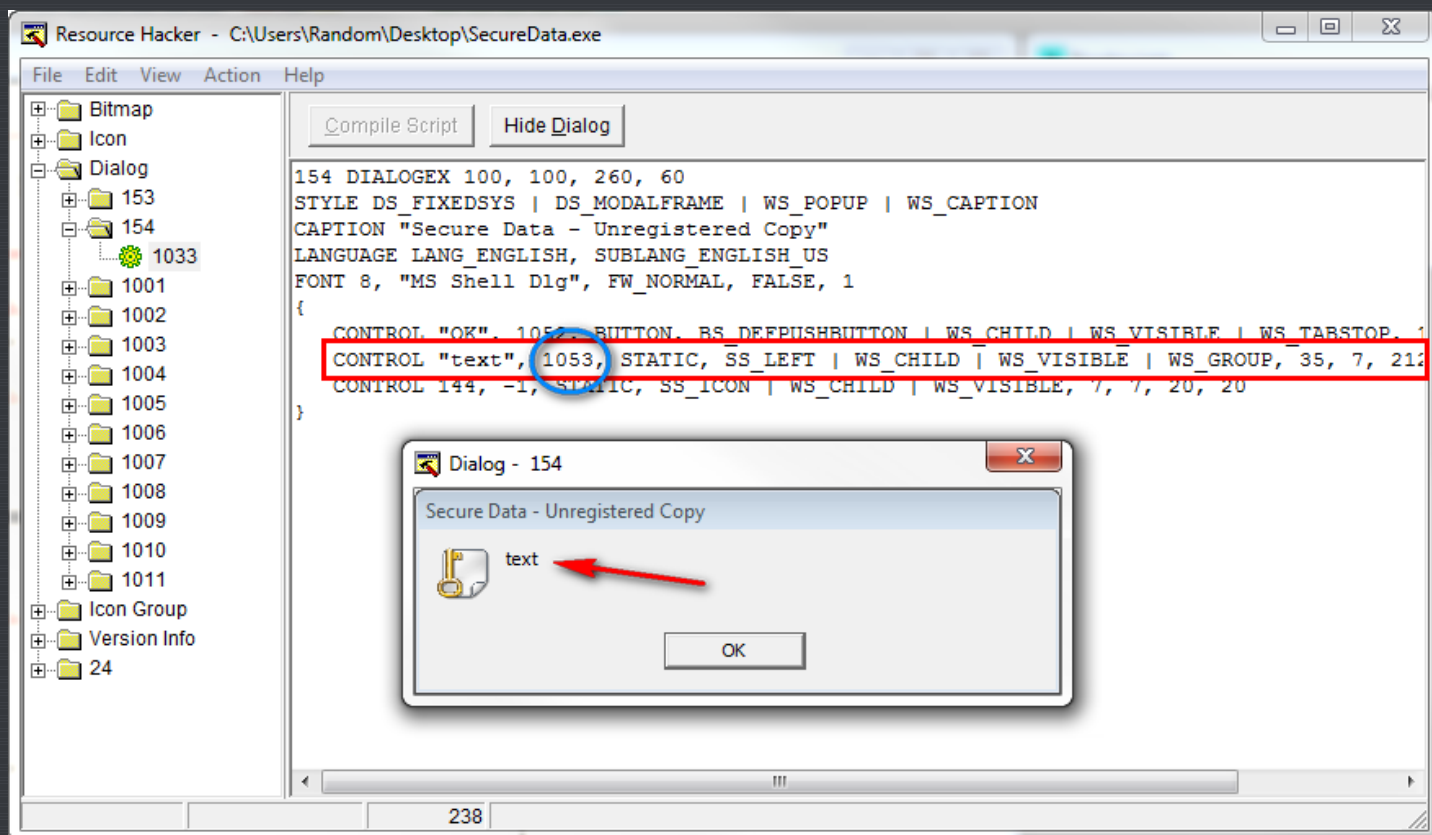
Another way we could deal with the time trial is to patch the code. This would be better as we would have to change the registry every time we used the app more than 256 times (0xFF is the largest number we could enter into that registry key without it taking 2 bytes, and crashing the app). Remember, when we first searched for strings we saw the string with the message that we had X amount of tries left. Let's go there and see what it looks like. First, we see the text being loaded at address 406078 (actually, a pointer to the text would be more correct):



Looking at this in a more big picture way, we can see that this is a routine in a Windows callback for when the WM_INITDIALOG message comes through:



The first thing it does is load a handle to a Windows control with an ID of 0x41D (1053) to be passed to the GetDlgItem API. Looking in Resource Hacker, we can see that ID 1053 corresponds to the text in the initial time trial nag dialog box:



Looking up GetDlgItem in the API help we see that this routine returns a handle to the Windows control in EAX. Setting a BP at address 40604C (the PUSH 41D instruction) and re-starting the app, then stepping down to this instruction, we see that the return value is 0x3E4 and is stored in EDI:

00406035	FF15 24724100	CALL DWORD PTR DS:[&USER32.EndPaint>]	EndPaint
00406038	E9 AE000000	JMP SecureDa.004060EE	
00406040	3935 ACEB4100	CMP DWORD PTR DS:[41EBAC],ESI	
00406046	0F85 A2000000	JNZ SecureDa.004060EE	
0040604C	68 1D040000	PUSH 41D	
00406051	56	PUSH ESI	
00406052	FF15 E8714100	CALL DWORD PTR DS:[&USER32.GetDlgItem>]	ControlID = 41D (1053.) hWnd = 000049C ('Secure Data - Unregistered Copy',o GetDlgItem
00406058	8BF8	MOV EDI,EAX	
0040605A	C74424 0C C48F4100	MOV DWORD PTR SS:[ESP+C],SecureDa.00418FC4	
00406062	C74424 10 00000000	MOV DWORD PTR SS:[ESP+10],0	
0040606A	C74424 68 00000000	MOV DWORD PTR SS:[ESP+68],0	
00406072	A1 84E04100	MOV EAX,DWORD PTR DS:[41E084]	
00406077	50	PUSH EAX	
00406078	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
0040607C	68 D0924100	PUSH SecureDa.004192D0	
00406081	51	PUSH ECX	
00406082	E8 C9E3FFFF	CALL SecureDa.00404450	UNICODE "You user32.75FDF212
00406087	007424 10	MOV ECX,DWORD PTR DS:[ESP+10]	
EAX=000B03E4 EDI=00000001			
Address	Hex dump	ASCII	
00417000	FE 40 0B 76 3F 77 0C 76 9D 46 0B 76 D6 14 0B 76	■@?w.v#Fv?l?v	
00417010	7D 15 0B 76 0D 46 0B 76 00 00 00 00 55 55 7D 00	■?w.v#Fv?l?v	
00418F434			000B66C 0018F438 0000001 0018F43C 00405F70 Se

The next statement loads the contents of address 418FC4 onto the stack at ESP+C. Following this constant in the dump, we see that that address stores another address, 403980, which if you follow, you will see is a callback. We can assume that this is the callback for the dialog box (for example, when you press OK):

Address	Hex dump	ASCII
00418FC4	80 39 40 00 54 00 65 00 6D 00 70 00 6F 00 72 00	C98.T.e.m.p.o.r.
00418FD4	61 00 72 00 79 00 46 00 69 00 6C 00 65 00 00 00	a.r.y.F.l.l.e...
00418FE4	54 00 65 00 6D 00 70 00 6F 00 72 00 61 00 72 00	T.e.m.p.o.r.a.r.
00418FF4	79 00 46 00 69 00 6C 00 65 00 2E 00 62 00 6D 00	y.F.i.l.l.e...b.m.
00419004	70 00 6D 00 50 00 72 00 65 00 70 00 61 00 72 00	D...P.r.e.D.a.r.
00419014	69 00 6E 00 67 00 00 00 64 00 61 00 74 00 61 00	I.n.g..d.a.t.a.
00419024	20 00 2E 00 2E 00 20 00 00 00 53 00 65 00 00 00S.e.
00419034	63 00 75 00 72 00 65 00 44 00 61 00 74 00 61 00	c.u.r.e.D.a.t.a.
00419044	00 00 00 00 45 00 6F 00 63 00 72 00 79 00 70 00	F.p.e.r.u.p

The next two lines load zeroes onto the stack (obviously initializing some local variables that will be used in the call) and then loads a very suspicious value into EAX from memory location 41E084:

00406040	> 3935 ACEB4100	CMP DWORD PTR DS:[41EBAC],ESI	Case 110 (WM
00406046	0F85 A2000000	JNZ SecureDa.004060EE	
0040604C	68 1D040000	PUSH 41D	ControlID =
00406051	56	PUSH ESI	hwnd = 00060
00406052	FF15 E8714100	CALL DWORD PTR DS:[&USER32.GetDlgItem>]	GetDlgItem
00406058	8BF8	MOV EDI,EAX	
0040605A	C74424 0C C48F4100	MOV DWORD PTR SS:[ESP+C],SecureDa.00418FC4	
00406062	C74424 10 00000000	MOV DWORD PTR SS:[ESP+10],0	
0040606A	C74424 68 00000000	MOV DWORD PTR SS:[ESP+68],0	
00406072	A1 84E04100	MOV EAX,DWORD PTR DS:[41E084]	
00406077	50	PUSH EAX	
00406078	8D4C24 10	LEA ECX,DWORD PTR SS:[ESP+10]	
0040607C	68 D0924100	PUSH SecureDa.004192D0	UNICODE "You
00406081	51	PUSH ECX	user32.75FDF
00406082	E8 C9E3FFFF	CALL SecureDa.00404450	
00406087	007424 10	MOV ECX,DWORD PTR SS:[ESP+10]	
DS:[0041E084]=00000008			
EAX=000B03E4			
Address	Hex dump	ASCII	
0041E084	00 00 00 00 01 00 00 00 00 00 00 00 06 07 03 00	■...0.....+..	
0041E094	C2 06 03 00 01 00 00 00 1C 07 03 00 00 00 00 00	T..0.....	
0041E0A4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0041E0B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0041E0C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0041E0D4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0041E0E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
0041E0F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Now, the reason I know it's suspicious is because this happens to be the exact value for the number of trials I have left (yours will be different). If you now restarted the app, this value would be one less, as you've used one of your trials. So here we can assume that this address, 41E084, stores the number of trials we have left. That is very helpful to know 😊.

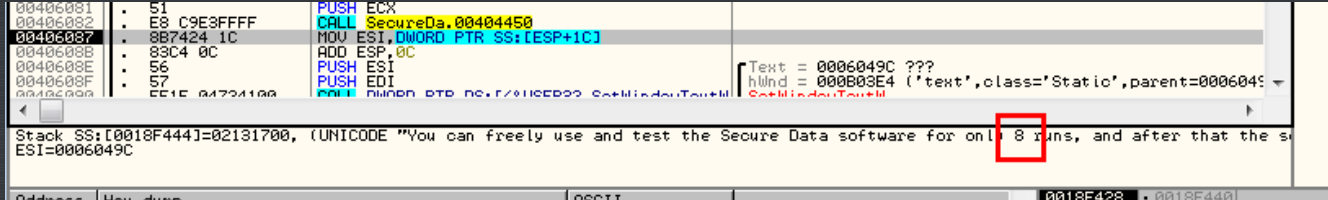
Finally, the target loads this value, as well as a pointer to the string, onto the stack and then makes a call. The reason for this call (at address 406082) is to check the value for the number of trials left (is it less than zero?) and to insert it into the string. You may have noticed that the string does not contain a value:

UNICODE "You can freely use and test the Secure Data software for only %d runs, and after that the software w"
user32.75FDF212

For you that have programming experience, you will recognize this as a printf type expression:

```
printf("My IQ is a whopping %d", 18);
```

Where the "%d" will be replaced with the decimal at the end of the expression. This call is doing the same thing. Since the number of trials is dynamic, we must create a generic string and then insert the actual number at runtime. Stepping past the call, we can see that it inserted the value into the string:



Now we know for sure that the address listed above does in fact contain the number of trials left... Finishing up this area of code, the time trial nag is simply displayed and waits for our clicking the OK button.

Patching the Target

Your first thought may be, "Why not just change the moving of the contents of that memory location (the one that has the number of trials left) from loading the number of trials left, to moving an arbitrary large number?" For example, why not change this instruction at address 406072:

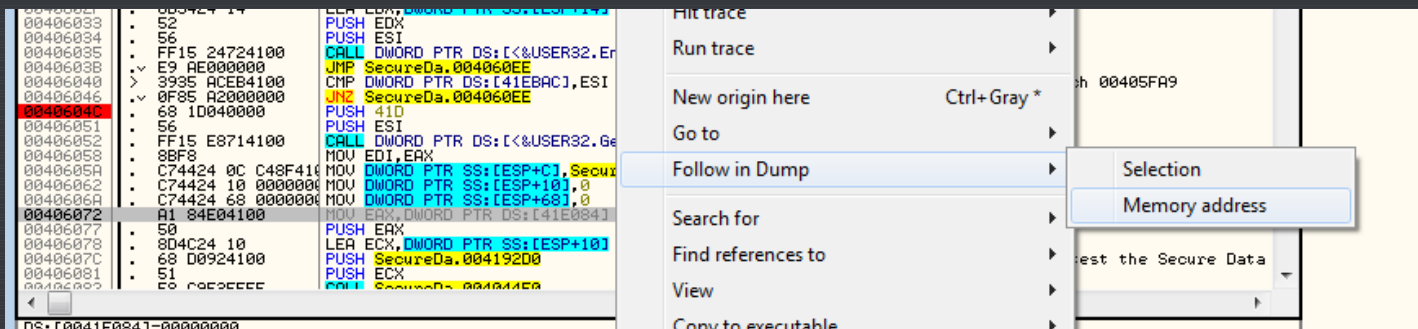
```
MOV EAX, DWORD PTR DS:[41E084]
```

to this:

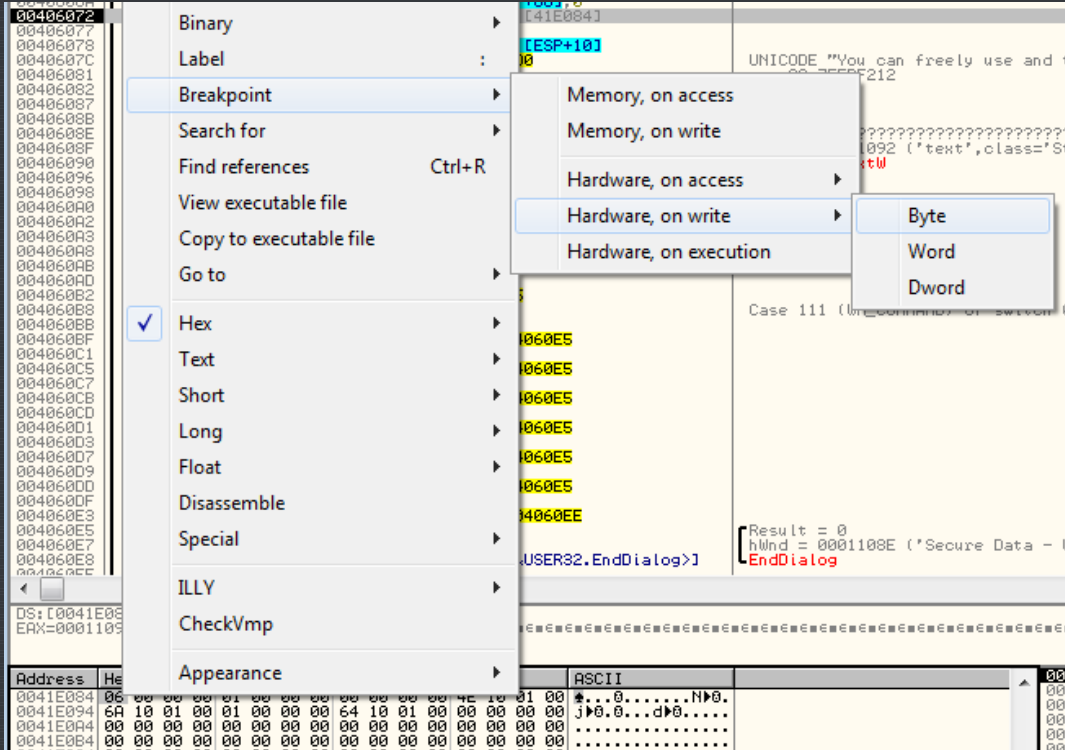
```
MOV EAX, 99
```

The reason is because all this would do is change the dialog...the actual check of the number of trials left is not bypassed. Therefore, when the number of trials is lower than 1, the app will stop working (even though the dialog will say we have 99 trials left). So what we have to do is find where the target loads this value from the registry and stores it into this variable.

The solution to this is to place a hardware breakpoint on this memory location, telling Olly to pause whenever a new value is stored in it. The first step is to follow this address in the dump:

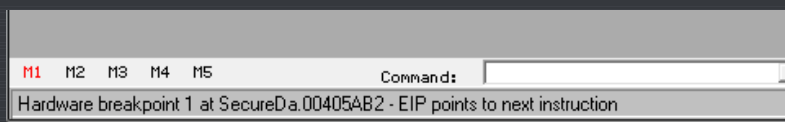


Then, in the dump, we want to place a hardware breakpoint on this location, pausing whenever a new value is written to it:

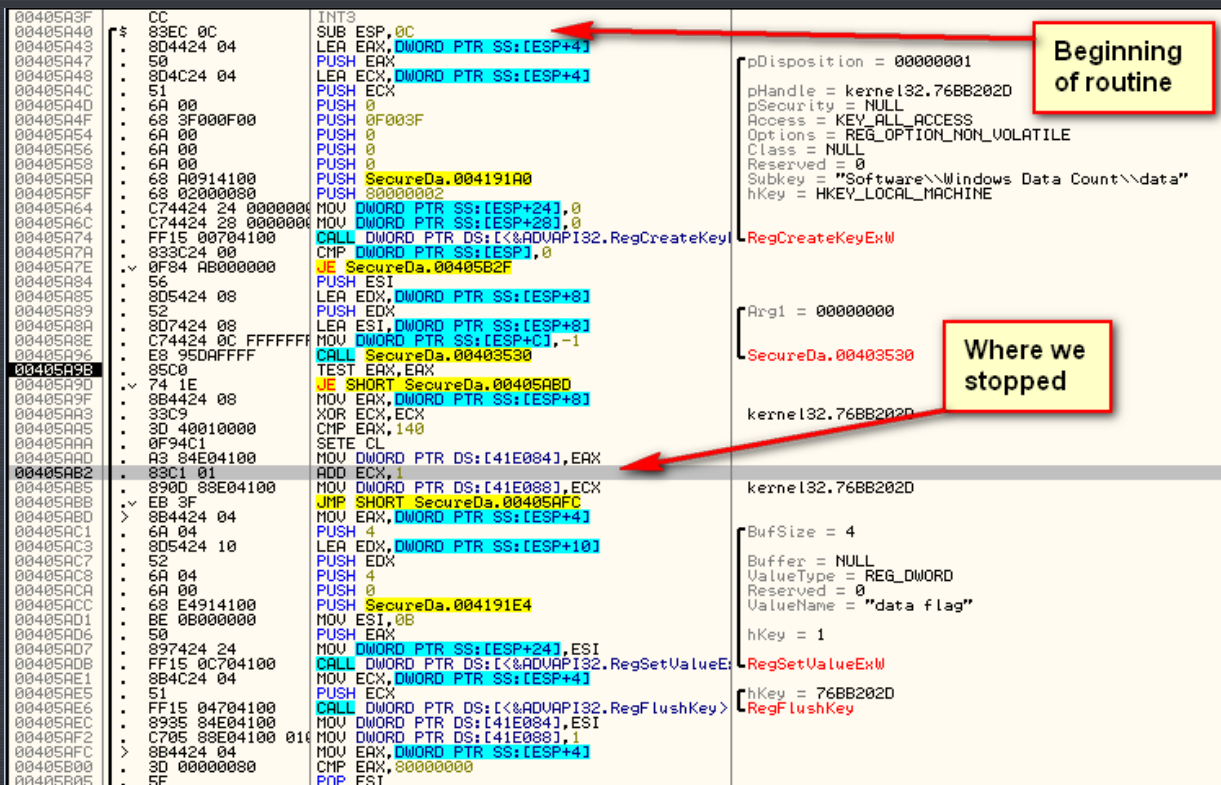


You can see that I have 6 trials left. We chose byte because that's the length of this variable. Now we have a HBP whenever there is a new value written to it.

Re-start the app now. Olly will break on our hardware breakpoint (HBP). You can tell by looking at the bottom of the Olly window:



Let's take a look at where we stopped:



At the top of the screen (in code prior to our breaking) we see where the registry value is pulled from the Windows Data Count folder (RegCreateKeyExW is used to not only create a key, but to open one as well). The target then does some processing on it at around address 405A7E, checking if the returned value is zero (meaning the target is not allowed to access the registry) and jumping to a badboy if it is (at 405B2F – telling us we need administrator privileges to access this key). If there's no error, the call at 405A96 actually loads in the value from that key, returning it in ESP+C (which will become ESP+8 after the return):

00403530	8B0E	MOV ECX, DWORD PTR DS:[ESI]	
00403532	83EC 08	SUB ESP, 8	<-- Attempt to open key
00403535	57	PUSH EDI	
00403536	8B3D 14704100	MOV EDI, DWORD PTR DS:[&ADVAPI32.RegQueryValueExW]	advapi32.RegQueryValueExW
0040353C	6A 00	PUSH 0	BufSize = NULL
0040353E	6A 00	PUSH 0	Buffer = NULL
00403540	8D424 0C	LEA EAX, DWORD PTR SS:[ESP+C]	
00403544	50	PUSH EAX	pValueType = 00000001
00403545	6A 00	PUSH 0	Reserved = NULL
00403547	68 E4914100	PUSH SecureDa.004191E4	ValueName = "data flag"
0040354C	51	PUSH ECX	hKey = 76BB2020
0040354D	FFD7	CALL EDI	RegQueryValueExW
0040354F	85C0	TEST EAX, EAX	<-- Is there an error?
00403551	74 09	JE SHORT SecureDa.0040355C	<-- No, so keep going
00403553	33C0	XOR EAX, EAX	<-- yes, so bug out
00403555	5F	POP EDI	SecureDa.00405A9B
00403556	83C4 08	ADD ESP, 8	
00403559	C2 0400	RETN 4	
0040355C	B8 04000000	MOV EAX, 4	
00403561	394424 04	CMP DWORD PTR SS:[ESP+4], EAX	<-- Wrong return type?
00403565	75 EC	JNZ SHORT SecureDa.00403553	<-- yes, so bug out
00403567	8B0E	MOV ECX, DWORD PTR DS:[ESI]	
00403569	8D5424 08	LEA EDX, DWORD PTR SS:[ESP+8]	<-- 18FE68
0040356D	52	PUSH EDX	
0040356E	894424 0C	MOV DWORD PTR SS:[ESP+C], EAX	<-- 18FE68 = length of buffer (4)
00403572	8B4424 14	MOV EAX, DWORD PTR SS:[ESP+14]	<-- 18FE70
00403576	50	PUSH EAX	<-- 18FE7C = buffer for return value
00403577	6A 00	PUSH 0	
00403579	6A 00	PUSH 0	
0040357B	68 E4914100	PUSH SecureDa.004191E4	UNICODE "data flag"
00403580	51	PUSH ECX	kernel32.76BB2020
00403581	FFD7	CALL EDI	
00403583	F7D8	NEG EAX	<-- 18FE7C = number trials left
00403585	1BC0	SBB EAX, EAX	
00403587	83C0 01	ADD EAX, 1	
0040358A	5F	POP EDI	SecureDa.00405A9B
0040358B	83C4 08	ADD ESP, 8	
0040358E	C2 0400	RETN 4	<-- # trials left = ESP + C
00403591	CC	INT3	

*** I have commented this code if you wish to investigate it further. ***

We then load the returned value into our variable at address 405AAD (my current value is 6). This is one instruction before we paused in Olly:

00405A98	85C0	TEST EAX, EAX	
00405A9D	74 1E	JE SHORT SecureDa.00405AB0	
00405A9F	8B4424 08	MOV EAX, DWORD PTR SS:[ESP+8]	
00405AA3	33C9	XOR ECX, ECX	
00405AA5	3D 40010000	CMP EAX, 140	
00405AA8	0F94C1	SETC CL	
00405AAD	8B 84E04100	MOV DWORD PTR DS:[41E084], EAX	
00405AB2	83C1 01	ADD ECX, 1	
00405AB5	83D0 38E04100	MOV DWORD PTR DS:[41E088], ECX	
00405AB8	EB 3F	JMP SHORT SecureDa.00405AFC	
00405ABD	8B4424 04	MOV EAX, DWORD PTR SS:[ESP+4]	
00405AC1	6A 04	PUSH 4	BufSize = 4
00405AC3	8D5424 10	LEA EDX, DWORD PTR SS:[ESP+10]	
00405AC7	52	PUSH EDX	Buffer = NULL
00405AC8	6A 04	PUSH 4	ValueType = REG_DWORD
00405AC9	6A 00	PUSH 0	Reserved = 0

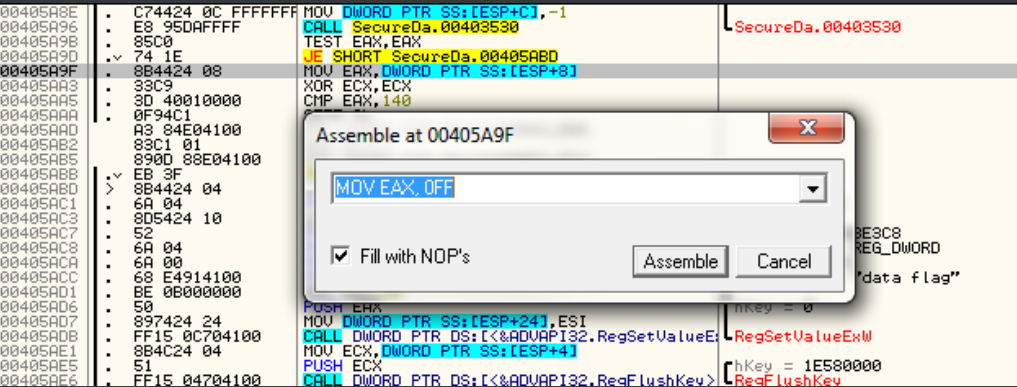
Finally, we check for some strange values, then close the key and return:

00405B00	3B 00000000	CMP EAX, 00000000	
00405B05	5E	POP ESI	
00405B06	74 45	JE SHORT SecureDa.00405B4D	
00405B08	3D 05000000	CMP EAX, 00000005	
00405B0D	74 3E	JE SHORT SecureDa.00405B4D	
00405B0F	3D 01000000	CMP EAX, 00000001	
00405B14	74 37	JE SHORT SecureDa.00405B4D	
00405B16	3D 02000000	CMP EAX, 00000002	
00405B18	74 30	JE SHORT SecureDa.00405B4D	
00405B1D	3D 03000000	CMP EAX, 00000003	
00405B22	74 29	JE SHORT SecureDa.00405B4D	
00405B24	50	PUSH EAX	hKey = 00000005
00405B25	FF15 08704100	CALL DWORD PTR DS:[&ADVAPI32.RegCloseKey]	RegCloseKey
00405B2B	83C4 0C	ADD ESP, 0C	
00405B2E	C3	RETN	
00405B2F	6A 30	PUSH 30	Style = MB_OK1MB_ICO

So the question becomes, where is the best place to patch the binary? Tracing back through the code, at address 405AAD the value that was received from the registry is stored into the memory location to check later on for the number of trials. Nothing has happened yet (no modifications) to this value- it has simply been loaded from the registry and stored here. The problem is, if you try patching this to something like MOV DWORD PTR DS:[41E084], FF, the code will be screwed up (it will delete the next two instructions):

00405AA3	33C9	XOR ECX, ECX	
00405AA5	3D 40010000	CMP EAX, 140	
00405AA8	0F94C1	SETC CL	
00405AAD	C705 84E04100 FF	MOV DWORD PTR DS:[41E084], 0FF	
00405AB7	90	NOP	
00405AB8	90	NOP	
00405AB9	90	NOP	
00405ABA	90	NOP	
00405ABB	EB 3F	JMP SHORT SecureDa.00405AFC	
00405ABD	8B4424 04	MOV EAX, DWORD PTR SS:[ESP+4]	
00405AC1	6A 04	PUSH 4	
00405AC3	8D5424 10	LEA EDX, DWORD PTR SS:[ESP+10]	
00405AC7	52	PUSH EDX	
00405AC8	6A 04	PUSH 4	

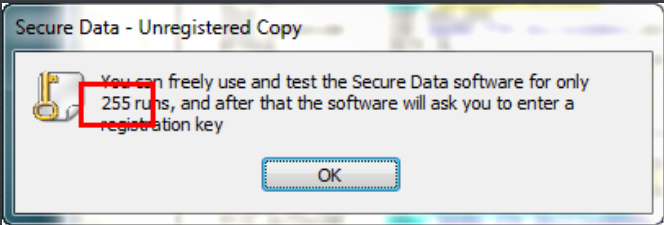
So, let's patch the previous instruction at 405A9F, where EAX is loaded with the returned number of trials:



And we can see, after stepping over it, that our variable now contains 0xFF instead of the real number of trials left:

Address	Hex dump
0041E084	FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E094	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0A4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0D4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E0F4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E104	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0041E114	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Go ahead and run the app:



You will obviously have to save the patch. If you don't know how, please see my previous tutorials. Once you do, you will always have 255 tries left, no matter how many times you run it.

Conclusion

Obviously, removing the nag altogether is a far better solution to this app, but I wanted to go over trials specifically for two reasons; 1) It's good to know how they work and to recognize them as they lead you to the actual protection scheme, and 2) because sometimes you cannot crack an app, and this is the next best thing 😊.

BONUS QUESTION: What are those strange looking checks for starting at address 405B08? And what is the check against 0x140 at address 405AA5 for?

-Till next time

R4ndom

No tags

Leave a Reply