REVERSE-ENGINEERING COMMERCIAL SOFTWARE

CASE STUDY: IonIce EXE LOCKER v.1.0

INTRODUCTION:

In this paper I will try to reverse engineer a shareware product that recently appeared on the market: IonIce EXE LOCKER. Its whole purpose is to protect executable from being run by an unauthorized person if that person doesn't have the password. In other words it's just another executable password protector.

I won't refer to the protection scheme of this little application, if you really are interested it's just ASProtect v.2.x, but I will mainly analyze the way that this software does its job: protecting your executables from other people.

We will analyze the stub of the stub (the code that handles password fetching and data decryption) and see the weak points and possible attacks that might be mounted against it. We'll also suggest some improvements that could be made.

PAPER:

You can this software from www.ionice.com.

The tools I will be using in this tutorial:

- IDA v.4.6.0.785 (that's my versin)
- OllyDebugger v.1.08d
- Hiew

Also you should have little cryptography knowledge like what a hashing algorithm is, what's a block cipher.

Ok before going into analysis let's develop our study plan:

- get as much information from just looking at the application and it's help file
- decide analysis method

Ok the application's interface is very easy to use and understand. It has a simple explorer type window in which you can look inside your hard drive and select the files you want to protect. You can protect an application, unprotect an application and change an application's password. And that's about all you can do with this soft.

Looking inside the help file you see:

“IonIce EXE Locker uses executable code encryption that prevents start-up of protected applications without the correct password. The protector uses many techniques which complicate password fingering, as well as debugging and disassembling protected programs.

Features:

- **Executable file password protection**
- **Encryption algorithm with 128-bit key length**
- **Dynamic encryption**
- **Anti disassembling techniques”**

Ok, the encryption algorithm is TEA (Tiny Encryption Algorithm), that I saw on the site of the application.

Right after reading this a question comes to mind: why the use of anti-debugging and anti-disassemble techniques if it just protects the executable from being ran? That doesn’t make too much sense: you either have the password and you can execute the executable or you purely can’t. If it protects it in the right way then you can’t just crack the loader. Well back to study….

Now, because the file is protected with ASProtect we can’t unpack it and study the code then (well we can but we won’t do it) so we do like this: we take an executable from our hard drive and protect it using IonIce EXE LOCKER and study the stub that it adds.

So get yourself any application that you want and protect it using our target. Also test that it works after being protected as this version of the application isn’t very stable and compatible.

Did you protect it?ok…

Now if you look inside with HIEW you’ll see that 2 new sections have been added “test” and “data” and that the entrypoint has been moved inside “test” section.

You also notice that it didn’t modify the import table (like adding a new import table).

Let’s go straight into analyzing it using IDA. So load the protected executable inside IDA and wait while IDA does its job.

Ok…

SO how does it work, well instead of pasting large amounts of disassembly from IDA I’ll just explain you the big picture. But keep in mind that with this paper you should have received the commented disassembly of the stub which I made for a program.Look over it for any questions.

So the executable starts running:

- it gets the address of the entrypoint by using the classical delta method:

```
.test:0043E000                call    $+5
.test:0043E005                pusha
.test:0043E006                mov     ebp, [esp+20h]
.test:0043E00A                sub     ebp, 5          ;
```

So ebp will contain the address of entrypoint. From now on all variables will be accessed using ebp and the distance between the address of that variable and the address of the entrypoint. This is what you call position independent code. Just like viruses use ☺

- saves all registers
- Gets Kernel32.dll base address using the well known return address trick. Let me explain. Each program that executes when it terminates must return to Kernel32.dll because he is the one the created/launched it. So knowing this you take the return address which is somewhere inside kernel and assuming that it' aligned on a 64kb boundary you search for 'MZ' and for 'PE' (IMAGE_DOS_HEADER.e_magic and IMAGE_NT_HEADERS.Signature). Yes another trick taken from virus coding.
- Once that it finds the base address of kernel it scans it's export table for GetProcAddress (the method used in this program is not the fastest nor the smallest, AUTHORS keep in mind this and improve it)
- Now that it has the address of GetProcAddress function it can find whatever function he wants. So next he gets all the apis the it needs:

```
aKernel32_dll     db 'KERNEL32.DLL',0
aUser32_dll  db 'USER32.DLL',0
aGetmodulehandlea db 'GetModuleHandleA',0
aExitprocess db 'ExitProcess',0
aHeapcreate  db 'HeapCreate',0
aHeapalloc   db 'HeapAlloc',0
aHeapfree    db 'HeapFree',0
aHeapdestroy db 'HeapDestroy',0
aLoadlibrarya     db 'LoadLibraryA',0
aFreelibrary db 'FreeLibrary',0
aGetprocaddress   db 'GetProcAddress',0
aDialogboxindirectp db 'DialogBoxIndirectParamA',0
aGetdlgitem  db 'GetDlgItem',0
aGetwindowtexta   db 'GetWindowTextA',0
aMessageboxa db 'MessageBoxA',0
aEnddialog   db 'EndDialog',0
aWsprintfa   db 'wsprintfA',0
```

- After that he allocates some memory from the Heap and copies there the resource for the dialog that asks for password.
- Next it calls DialogBoxIndirectParamA
- After returning from there it frees the Heap, it frees user32.dll and destroys the heap.
- A variable is checked if set or not and if no set then it goes to the original entrypoint and executes the application else it just exits using ExitProcess.

Ok, not let's see the interesting part: the Window handling procedure used by DialogBoxIndirectParamA

- It handles only one message: WM_COMMAND for the OK button (that explains why when you press the close window button it doesn't close)
- When OK button is pressed it takes the handle of the editcontrol (GetDlgItem) and then it takes the password.
- The password is hashed using classical MD5

- Using the hash obtained earlier it decrypts a 32bytes ciphertext
- The decrypted ciphertext is then hashed with MD5 and the hash is compared with a hard coded one. If they match the 32 decrypted bytes are copied at the start of the OEP (Original Entry Point) and success flag is set to 0 (success). Else it just shows the Bad Password message box and sets the flag to 1.(failure).

That concludes our analysis. So let's resume how the password protection is done:

- select a file with no overlays (read inside help for info)
- get password to use when protecting
- get first 32bytes from the executable, to protect, entrypoint and hash them using MD5 and keep the hash
- hash the password entered by user with MD5 and use the obtained hash (128bits) to encrypt the first 32 bytes from entrypoint using TEA. Keep the resulted ciphertext for later decryption (inside stub)
- Set the original 32bytes from entrypoint to 0

So when you enter a password to execute the protected executable:

- Your password is hashed with MD5
- Password is used to decrypted the 32bytes ciphertext
- The decrypted bytes are hashed using Md5 and the obtained hash is compare to the hardcoded one (made when protecting executable)
- If they match then decrypted bytes are copied at OEP and then execution continues there.

Now let's study it from coders point:

- using this method the coder saved him the trouble of coding a full pe-crypter: let me explain. If he encrypted almost all the application (all section,resources) then he would have to code a proper decrypter too: which mean coding a IAT rebuilder, TLS handler, resource decrypter which would have meant allot more trouble and allot less compatility. Anyone who has tried coding a PE-Crypter knows what pain it involves.
- Using this method the coder can easily implement the Unprotect and the Change Password feature (for the second just replace the 32 encrypted bytes with other 32 encrypted bytes, this time encrypted with the md5 hash of the new password).
- Another thing: the application is decrypted allot faster, because you just decrypt 32 bytes instead of almost 90% of the application as you would have done if using the PE-Crypter method.

Now let's see method that would enable use to successfully run a protected file without knowing the password.

Let's see the attacks:

Dictionary attack:

You get a dictionary file (a file containing most used words from the English or any other language) and you try to decrypt the 32 bytes using the words inside till you get lucky. But this attack is universal. The fun fact is that it's the more effective: because most of the people put quite short passwords that they can remember, passwords that are common words.

Compiler Specific Attack:

No, it's nothing that you will find if studying cryptanalysis dedicated papers.
It's more of a silly name given by myself. But this attack would more likely to have more then 20– 30% success rate in the wild, well if the attacker has some knowledge of reverse engineering. But that too can be overtaken by making an automatic tool to do the task for you.
Let me explain what this attack is base on: just like I explained before the Exe Locker only encrypts the first 32 bytes of the application. Ok, but those 32 bytes are taken directly from the entrypoint.
This is the weak point: every compiler when compiling an application it puts at entrypoint some known bytes which are common for all application build with it. This bytes are just the code that compiler generates for the initialization of the application.

For example:
[Microsoft Visual C++ v5.0/v6.0]
558BEC6AFF68::::::::68::::::::64A1000000005064892500000000083::::

":" – represents a hex character like 0x0,0x1,…,0xF
"::" –represents a byte 0x00,0x01…,0xFF
So that :: could be any byte.
So just like you saw if an application was made in MSVC 5.0 or 6.0 then we have 22 fixed bytes from 32 needed. So now our task is to find the remaining 10 bytes
Well let me show you how you find another 4 bytes:
Launch your MSVC 6.0 or 5.0 and make a win32 application with no MFC which just shows a message box. Could be your typical HELLO WORLD first application and compile it under with DEBUG configuration and load it in olly:

```
00416D90 >/$ 55             PUSH EBP
00416D91  |. 8BEC           MOV EBP,ESP
00416D93  |. 6A FF          PUSH -1
00416D95  |. 68 68E34300    PUSH RAT.0043E368
00416D9A  |. 68 64C04100    PUSH RAT.__except_handler3
00416D9F  |. 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00416DA5  |. 50             PUSH EAX
00416DA6  |. 64:8925 000000>MOV DWORD PTR FS:[0],ESP
00416DAD  |. 83C4 A4        ADD ESP,-5C
```

So these are exactly our first 32 bytes.
But what's that except_handle3 go with olly at it's address:

```
0041C064 >/$ 55              PUSH EBP                                ;
Structured exception handler
0041C065  |. 8BEC            MOV EBP,ESP
0041C067  |. 83EC 08         SUB ESP,8
0041C06A  |. 53              PUSH EBX
0041C06B  |. 56              PUSH ESI
0041C06C  |. 57              PUSH EDI
0041C06D  |. 55              PUSH EBP
0041C06E  |. FC              CLD
0041C06F  |. 8B5D 0C         MOV EBX,DWORD PTR SS:[EBP+C]
0041C072  |. 8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0041C075  |. F740 04 060000>TEST DWORD PTR DS:[EAX+4],6
0041C07C  |. 0F85 82000000   JNZ RAT.0041C104
0041C082  |. 8945 F8         MOV DWORD PTR SS:[EBP-8],EAX
0041C085  |. 8B45 10         MOV EAX,DWORD PTR SS:[EBP+10]
0041C088  |. 8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
```

Well it seems it's the exception handler set up by MSVC in case any exception arises. This exception handler has the same code for all MSVC executables compiled without MFC (MFC is a little different). So we can use several bytes from the code (like 558BEC83EC0853565755FC which is more then enough) to search inside our file we want to execute without password, for the address of except_handle3.

This is how you get 4 more bytes clear.

There are still remaining 6 bytes. Let's see.

Now Load the same executable in IDA:

```
.text:00416D90 55              push    ebp
.text:00416D91 8B EC           mov     ebp, esp
.text:00416D93 6A FF           push    0FFFFFFFFh
.text:00416D95 68 68 E3 ..     push    offset stru_43E368
.text:00416D9A 68 64 C0 ..     push    offset __except_handler3
.text:00416D9F 64 A1 00 ..     mov     eax, large fs:0
.text:00416DA5 50              push    eax
.text:00416DA6 64 89 25 ..     mov     large fs:0, esp
.text:00416DAD 83 C4 A4        add     esp, 0FFFFFFA4h
```

Yes , it's the same code as in olly, but now double click on stru_43E368

```
.rdata:0043E368 FF FF FF FF        stru_43E368    dd 0FFFFFFFFh        ; _unk
.rdata:0043E368
.rdata:0043E368                    dd offset loc_416EAF    ; FilterProc
.rdata:0043E368                    dd offset sub_416ECA    ; ExitProc
.rdata:0043E374 60 F8 43 00        dd offset ??_R4type_info@@6B@ ;
```

So let's keep in mind that the first dword at that address is 0xFFFFFFFFh.

And two dwords following it contain addresses quite close to each other and always < 0xFF (the maximum value is much smaller but just to be sure).

The last two bytes can be very easily brute forced. One more thing to keep in mind

83 :: ::

|

this byte is either 0xC4(for addition) or 0xEC(for subtraction)

So you only have to brute 512 possibilities. Well ain't that much better then having to brute 2^128 possibilities ☺

Let's recap what you have to do if you know your protected executable was compiled with MSVC:

- Get the OEP address from the protected executable
- Search for except_handler3 addresss (search for these bytes 558BEC83EC0853565755FC)
- search inside the executable for the following: 0xFFFFFFFF,A,B where A,B are dwords with B – A < 0xFF and both A and B point inside the first section of the executable (the code section)
- brute force the other 2 missing bytes
- write the correct bytes at the start of the executable
- replace the entrypoint address from the PE header with the address obtained on step 1.

The bad news is that this works only for MSVC 5.0 and 6.0 without MFC.

But the procedure can be applied almost the same for any known compiler (depending on how many static bytes you have)., but like I said it requires quite allot of work to find something that helps you finding the missing bytes. You have to find connections and study that compiler quite a little.

Another question arises: before applying a procedure for the specific compiler and version how do you know what compiler it is?

There are some options:

- if the executable has resources study the resources, if you see some RC_DATA type of resource it's Delphi.
- Study the OEP code of the target after the missing 32 bytes. They will help you quite a little.

You have to do it so you should have to brute at maximum 2^32 variants any larger number isn't practical (that depends on your need to have that executable decrypted).

In conclusion I might say that executables protected with this application are very weak in front of a dedicated attacker, so I would suggest the author to work on the protection. I'm always opened to help if they want my help or suggestions.

FINAL WORDS:

Reverse-engineering is not only about destruction of protection schemes, but about construction of better protections, about making code better, solving problems and most of all improving those little 0's and 1's that make our day ☺.

Regards,

bLaCk-eye - bLaCk@reteam.org