ARTeam

the international bank of knowledge

2009

Defeating the Winlicense Main Executable version 2.0.5.0

Quo sego

ARTeam

Jan 2009

DISCLAIMER

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this tutorial have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched by other fellows, and cracked versions were available since a lot of time. ARTeam or the authors of the papers shouldn't be considered responsible for damages to the companies holding rights on those programs. The scope of this document as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

ARTEAM QUALITY 100% GUARANTEED

TABLE OF CONTENTS

FOREWORDS

This paper will discuss 2 protections only used in the Winlicense Main executable.

The protections discussed will be:

- CryptoCode (trivial name)
    - (Winlicense using threads to decrypt/encrypt certain code functions)
- Dll Database & LoadLibrary API
    - (Winlicense using an encrypted dll database, and a modified LoadLibrary API)

In this paper I will not discuss other aspects of the Winlicense protection scheme unless required to understand the above.

Defeating the above and the standard Winlicense protection options as used in commercial programs, will result in a functioning Main unpacked executable. However the protected apps will not function since the VM mutation/creation engine also checks for protection integrity. But not to spoil you guys you may find that one yourself. :)

For this Tutorial I've used the Winlicense.exe provided by hacnho. Thank you.

quosego

1 CRYPTOCODE

This protection is quite similar to CodeEncrypt, however it's a lot more complex. Defeating it however is even easier than defeating CodeEncrypt. Like CodeEncrypt it first decrypts a function executes it and then re-encrypts it.

First I'll show you a call to the protection:

```
006043F5        68 45382678        PUSH 78263845     (1)
006043FA        6A 01              PUSH 1            (2)
006043FC        6A 00              PUSH 0            (3)
006043FE        68 A41C2D61        PUSH 612D1CA4     (4)
00604403        68 6E857FE8        PUSH E87F856E     (5)
00604408        68 45382678        PUSH 78263845     (6)
0060440D                           CALL 00408C9C     (7)
                E8 8A48E0FF
```

As you can see it looks like a reasonably normal call and a few pushes, and if you would follow the call to 408C9C(1) you'll see that it'll just nicely go to the delphi API table. Fixing the direct API's here using the methods available on the RE boards would make it point to wsprintfA. Yet the pushes accompanying this call do not suggest a wsprintfA call, on the contrary the wsprintfA would actually crash if you'd make these pushes.

So did Oreans update the API writing routines in packer code and make the method used in the available tuts invalid? No, the wsprintfA API is actually correct. However not exclusively. If you don’t fix this API call you’ll see that the direct jump behind this call actually points to the Winlicense section instead of normally to a memory buffer containing (part of) the obfuscated API.

It points to the following function executed in the Main Thread:
I'll discuss each part of it. Execution is sequential.

1.1 MAIN THREAD

```
52      PUSH EDX
8BD4    MOV EDX,ESP
60      PUSHAD
```

1.1.1 START OF THE FUNCTIONS/STORAGE OF REGISTERS

```
E8 00000000     CALL 01613299
5D              POP EBP
81ED FCC06409   SUB EBP,964C0FC
```

Load EBP, EBP is used in retrieving fixed memory values. Intriguingly this is done with a call that calls the next line which then pops the call into EBP. Essentially moving 01613299 into EBP, a value is then subtracted to get the required ebp value as the acquired value from this call/pop method of course depends on the location it is used.

```
8B42 08        MOV EAX,DWORD PTR DS:[EDX+8]
3D 45382678    CMP EAX,78263845
0F85           JNZ 016133E6
```

```
38010000
```

This checks for the last instruction pushed, is it 78263845? In our case it is. However if it's not it'll branch of to the following instruction:

```
61             POPAD
5A             POP EDX
B8 ADA8397E    MOV EAX,USER32.wsprintfA
FFE0           JMP EAX
```

Hmmm, as you can see it jumps to wsprintfA if 78263845 is not pushed. Meaning that this function has two functions both to execute wsprintfA and decrypt a function.

```
8B42 0C         MOV EAX,DWORD PTR DS:[EDX+C]       //push value (2)
8B4A 18         MOV ECX,DWORD PTR DS:[EDX+18]      //push value (5)
D3C8            ROR EAX,CL
BB FEFD5F74     MOV EBX,745FFDFE
33C3            XOR EAX,EBX                        // Eax will hold the location.
83E8 04         SUB EAX,4
8985 C9282C09   MOV DWORD PTR
                SS:[EBP+92C28C9],EAX
8B58 04         MOV EBX,DWORD PTR DS:[EAX+4]
8BF8            MOV EDI,EAX
```

Here it uses two pushed values (5) & (2) to calculate the location of the encrypted function.
And next store it, and use it for the next function.

```
8B42 10         MOV EAX,DWORD PTR DS:[EDX+10]          //push value (4)
8B4A 18         MOV ECX,DWORD PTR DS:[EDX+18]          //push value (5)
D3C8            ROR EAX,CL
BE BECDF630     MOV ESI,30F6CDBE
33C6            XOR EAX,ESI                           // Eax will hold the end location
                                                      of the encrypted function.
2BC7            SUB EAX,EDI
83E0 FC         AND EAX,FFFFFFFC
83E8 04         SUB EAX,4
8985 3D242C09   MOV DWORD PTR SS:[EBP+92C243D],EAX
8B42 14         MOV EAX,DWORD PTR DS:[EDX+14]          //retrieve 0 dword. (3)
8985 F1192C09   MOV DWORD PTR SS:[EBP+92C19F1],EAX
8B42 18         MOV EAX,DWORD PTR DS:[EDX+18]
```

Here it uses two pushed values (5) & (4) to calculate the end location of the encrypted function.
Afterwards it stores it and resets eax. It also stores the 0 dword pushed (3) at a memory location. This is a pointer to make the function encrypt or decrypt. 0 = decrypt, 1 = encrypt.

```
53                  PUSH EBX
```