“...And a pinch of Delphi”

Originally published on 2003-10-30

Getting an overview

Attachment, genme.exe, 806912 bytes, 491E6EE050644FF76F0FC05E8F2683F5

Since our goal is to write a keygen for this target that is really what we should be focusing on. However, the crackme's readme file already mentioned two strategies used in the protection, namely anti-debugging and integrity checking through CRC. I thought we'd have a look at these first because the anti-debugging might be interfering with our examination of the serial algorithm.

There are two basic ways for a binary to perform integrity checks on itself. It calculates a checksum from either the disk image or from the process image, which it compares to a good checksum. In the case of our target it uses the CRC32 algorithm on part of the disk image. We are in luck here, because to calculate the checksum of a disk image the process has to somewhere request a handle to the file to be able to access the data in it. A few protections use the obsolete _lopen() function exported from kernel32.dll to open files. This is mostly because modern (compiled) binaries wouldn't normally use that function and thus the reverser might not think of looking for it. Our target, however, uses the normal and much expected CreateFileA(), also a function residing in kernel32.dll.

To find the routine that does integrity checking, load our target into a debugger and put a breakpoint on CreateFileA(). Whenever you're using software breakpoints I recommend disassembling the API quickly and putting the actual breakpoint a few instructions into the API. This is because some protections might be checking the API's entry point to make sure it is clean from breakpoints.

Once the breakpoint triggers view the stack to see what parameters were passed to the function. In this case we need the ASCII string passed to CreateFileA() to be the full path and filename of our target. Step out and you will be smack in the middle of the routine we were looking for.

The routine that performs integrity checking starts at VA 454784. I won't show it in its entirety because the code is very clean and easy to follow. The disk image is first memory mapped before the actual CRC32 routine takes over:

```
00454840  MOV   EDX, EBP                                  // *data
00454842  LEA   EAX, DWORD PTR SS:[ESP+4]                 // *sum
00454846  MOV   ECX, DWORD PTR DS:[45BC5C]                // length
0045484C  CALL  004548B0_CRC32_Update
```

CRC32_Update is a table-driven implementation of CRC32 that supports operating on streaming data. The reason for using two streams of data is simple; if you were to calculate a checksum for the file containing the checksum you could never synchronize it. Whenever you have the correct checksum and write it back to the binary you automatically invalidate it. In this implementation a small block of data (0x24 bytes long) from the PE "DATA" section is excluded from the calculations. One thing that struck me as odd is the initial value of the "sum" variable. Normally you use a long with all bits set (0xFFFFFFFF), but instead a value of 0xFFF00FFF is being used.

The code responsible for integrity checking is in fact a component, TOgProtectExe, which is part of the TurboPower OnGuard package.

The critical point where the checksum gets verified is at VA 454878. If the checksum is found to be invalid the application exits silently after a while. This is at VA 452ACF, using PostQuitMessage(). It is, by the way, the same routine used for exiting from the anti-debugging code.

The anti-debugging routine is executed once during start-up but the interesting thing is that a timer is used for also executing it repeatedly. The timer object is based on "TPUtilWindow" and there are two timers all in all in this application; one for the anti-debugging and another one for on-mouse-over effects on the buttons. Delphi has set it up so that a common top-level message dispatcher at VA 41CD10 connects to all window procedures. This approach works well because all windows in the application have been instantiated from their respective templates, allowing the dispatcher to branch smoothly. The "TPUtilWindow" windows share the same basic window procedure at VA 42B7D0 because they are both timers. The anti-debugging timer is set up using SetTimer() and has a sleep count of 50 milliseconds between each time it triggers. The only message that is filtered out and acted upon is WM_TIMER, all others are passed on to DefWindowProcA(). Every once in a while the message is WM_TIMER and execution can be seen worming its way down through the layers, finally arriving at VA 45A554. We have found the anti-debugging routine! Here's a snippet of the effective code:

```
0045A592  PUSH   EAX                                      // *symbol
0045A593  PUSH   EBX
0045A594  CALL   GetProcAddress
0045A599  MOV    EDI, EAX
0045A59B  MOV    ESI, EDI
0045A59D  TEST   EDI, EDI                                 // win98+ ?
0045A59F  JZ     0045A5A7_abort
0045A5A1  CALL   ESI
0045A5A3  MOV    EBX, EAX
0045A5A5  JMP    0045A5CF_finale
```

GetProcAddress() is used to dynamically retrieve the address of IsDebuggerPresent(). If the return value is 0, meaning the function is unknown and not exported from the suggested library, the routine will abort. This happens in Windows 95, Windows NT 3.5 etcetera, relieving users of these operating systems from the anti-debugging.

A few lines up you will find something a little more interesting:

```
0045A578  LEA    ECX, DWORD PTR SS:[EBP-4]                   // **buffer
0045A57B  MOV    EDX, 0B9                                    // xor key
0045A580  MOV    EAX, 0045A614                               // *string
0045A585  CALL   0045A4B8_Decrypt_String
```

You might have attempted to extract all strings in our target and noticed that some strings available at run-time could not be found in the binary. That's because all important strings were encrypted at compile-time. The good news is that there is only one function used for decrypting the strings. And we know where it is.

The anti-debugging code can be observed decrypting two strings, "IsDebuggerPresent" and "GetProcAddress". We can find the encrypted versions of these strings at VA 45A614 and VA 45A630, respectively. A reasonable guess is that 0045A4B8_Decrypt_String will lead us to the serial algorithm. I propose a conditional breakpoint that breaks only when a string other than those already seen, is being decrypted. Something like: (when) EAX != 45A614 && EAX != 45A630.

The serial algorithm

Using the above conditional breakpoint technique you will find the serial routine and can confirm that it starts at VA 45A7C4. One thing you will see when working with Delphi applications is a common gateway for communicating with and controlling the application's windows. In this snippet 00432F0C_Get_Text wraps around it:

```
0045A7E4  LEA    EDX, DWORD PTR SS:[EBP-4]                   // **buffer
0045A7E7  MOV    EAX, DWORD PTR DS:[EBX+2FC]                 // object
0045A7ED  CALL   00432F0C_Get_Text
0045A7F2  MOV    EAX, DWORD PTR SS:[EBP-4]                   // *buffer
0045A7F5  CALL   004045D4_Pascal_Strlen
0045A7FA  TEST   EAX, EAX
0045A7FC  JLE    0045A8A6_abort
```

Tracing into 00432F0C_Get_Text, you will eventually end up at VA 437254 where you can see the actual directions/requests being sent using CallWindowProcA(). When reading the text of a window, the WM_GETTEXTLENGTH message is first sent and shortly thereafter follows a WM_GETTEXT message.

The code that does the actual serial generation is located in a subroutine starting at VA 45A640. Important parts of the code include:

```
0045A66B  XOR EBX, EBX

...

0045A67A  MOV    DWORD PTR SS:[EBP-8], EAX         // name length + 1
0045A67D  XOR    EDI, EDI
```

```
0045A67F  MOV    EAX, DWORD PTR SS:[EBP-4]         // pointer to name

0045A682  MOVZX  EAX, BYTE PTR DS:[EAX+EDI-1]      // grab character
0045A687  ADD    EAX, EAX                          // multiply by 2

0045A689  ADD    EBX, EAX                          // add to total
0045A68B  MOV    EAX, DWORD PTR SS:[EBP-4]
0045A68E  CALL   004045D4_Pascal_Strlen
0045A693  SUB    EAX, EDI
0045A695  MOV    EDX, DWORD PTR SS:[EBP-4]
0045A698  MOVZX  EAX, BYTE PTR DS:[EDX+EAX-1]      // grab character
                                                   // from the right
0045A69D  MOV    ECX, 3
0045A6A2  XOR    EDX, EDX
0045A6A4  DIV    ECX                               // divide char
                                                   // value by 3

0045A6A6  ADD    EBX, EAX                          // add to total

0045A6A8  INC    EDI                               // string index
0045A6A9  DEC    DWORD PTR SS:[EBP-8]
0045A6AC  JNZ    0045A67F                          // loop?


0045A6AE  XOR    EBX, 1                            // switch parity
0045A6B1  SHR    EBX, 3                            // divide total by 8
0045A6B4  MOV    EAX, DWORD PTR SS:[EBP-4]
0045A6B7  CALL   004045D4_Pascal_Strlen
0045A6BC  SHL    EAX, 2                            // multiply length
                                                   // of name by 4

0045A6BF  ADD    EBX, EAX                          // add to total
```

Note that the code at VA 45A682 operates out of bounds, reading "before" the string. So does the code at VA 45A698. The byte read is in practice always zero because of the way string objects are stored in Pascal/Delphi. Also note that the characters in the entered name are treated as unsigned.

The next couple of lines and the routines called perform a "printf" and more. The value of 'total' (treated as signed) is first written out as a string in decimal format. It is then truncated at 5 characters. If the string is less than 5 characters wide it is padded on the left with "0" until it is exactly 5 characters wide. Moving on beyond that you will see:

```
0045A715  MOV    EAX, DWORD PTR SS:[EBP-4]         // pointer to name
0045A718  CALL   004045D4_Pascal_Strlen
0045A71D  SAR    EAX, 1                            // divide by 2
0045A71F  JNS    0045A724
0045A721  ADC    EAX, 0
```

Since the length of the name will never have its 32nd bit set, it can never be interpreted as signed, and the SAR in effect does a division by two.

```
0045A724  TEST   EAX, EAX
0045A726  JNZ    0045A733                          // jump is taken
                                                   // if name is
                                                   // two characters
                                                   // or longer
```

```
0045A728  MOV   EAX, ESI
0045A72A  MOV   EDX, DWORD PTR DS:[ESI]
0045A72C  CALL  00404370_HLL_management
0045A731  JMP   0045A75D

0045A733  MOV   EAX, DWORD PTR SS:[EBP-4]        // pointer to name
0045A736  CALL  004045D4_Pascal_Strlen
0045A73B  SAR   EAX, 1                           // divide by 2
0045A73D  JNS   0045A742
0045A73F  ADC   EAX, 0

0045A742  MOV   EDX, DWORD PTR SS:[EBP-4]
0045A745  MOV   DL, BYTE PTR DS:[EDX+EAX-1]      // grab character
0045A749  LEA   EAX, DWORD PTR SS:[EBP-18]       // **temp_buffer
0045A74C  CALL  004044FC_Store_Character
0045A751  MOV   EDX, DWORD PTR SS:[EBP-18]       // *temp_buffer
0045A754  MOV   ECX, DWORD PTR DS:[ESI]          // *serial
0045A756  MOV   EAX, ESI                         // **serial
0045A758  CALL  00404620_Concatenate_Strings
```

The serial is appended to character from temp_buffer, thereby forming a new serial.

```
0045A75D  MOV   EAX, DWORD PTR SS:[EBP-4]        // pointer to name
0045A760  CALL  004045D4_Pascal_Strlen
0045A765  MOV   EDX, DWORD PTR SS:[EBP-4]
0045A768  MOV   DL, BYTE PTR DS:[EDX+EAX-1]      // grab last char
0045A76C  LEA   EAX, DWORD PTR SS:[EBP-1C]       // **temp_buffer
0045A76F  CALL  004044FC_Store_Character
0045A774  MOV   EDX, DWORD PTR SS:[EBP-1C]       // *temp_buffer
0045A777  MOV   EAX, ESI                         // **serial
0045A779  CALL  004045DC_Concatenate_Strings
```

Character in temp_buffer is appended to the serial number.

```
0045A77E  LEA   EDX, DWORD PTR SS:[EBP-20]       // **final_serial
0045A781  MOV   EAX, DWORD PTR DS:[ESI]          // *serial
0045A783  CALL  004082EC_Uppercase_String
```

Reflections

...All too many to make sense. I'll leave you with just one more observation: the text on the main form is not scrolled using a timer. The latency introduced in scrolling is caused by invoking the Sleep() API function. You should be able to confirm this by putting a breakpoint on CreateThread() as you start the target.

You may contact the author of this short essay via sna@reteam.org