Information Security Lab

Assignment-2

Manish Choudhary(GTID: 902982487) - September 28, 2014

1. Describe in detail how the program's runtime-generated code is created. What is the mechanism used to change data into code? What is the virtual address range of the code that performs this transformation?

The code initially (when the data section is accessed for the first time) performs some checks and modifies some bytes in the data section. But, those bytes are not used as part of dynamically generated code.

The actual dynamic code generation process starts at:

```
004011FB  |. C745 F8 000000>MOV DWORD PTR SS:[EBP-8],0
```

This initializes a local variable to 0 which will be used to keep track of the number of iterations and also as an offset which will be added to a base address to access the bytes in the data section.

The dynamic code gets generated by following loop:

```
00401202  |> 837D F8 3D        /CMP DWORD PTR SS:[EBP-8],3D
00401206  |. 76 08             |JBE SHORT 1f992c83.00401210
00401208  |. EB 2B             |JMP SHORT 1f992c83.00401235
0040120A  |  8DB6 00000000     |LEA ESI,DWORD PTR DS:[ESI]
00401210  |> B8 08204000       |MOV EAX,1f992c83.00402008
00401215  |. 8B55 F8           |MOV EDX,DWORD PTR SS:[EBP-8]
00401218  |. 8A0402            |MOV AL,BYTE PTR DS:[EDX+EAX]
0040121B  |. 8845 F7           |MOV BYTE PTR SS:[EBP-9],AL
0040121E  |. 8075 F7 7F        |XOR BYTE PTR SS:[EBP-9],7F
00401222  |. B8 08204000       |MOV EAX,1f992c83.00402008
00401227  |. 8B55 F8           |MOV EDX,DWORD PTR SS:[EBP-8]
0040122A  |. 8A4D F7           |MOV CL,BYTE PTR SS:[EBP-9]
0040122D  |. 880C02            |MOV BYTE PTR DS:[EDX+EAX],CL
00401230  |. FF45 F8           |INC DWORD PTR SS:[EBP-8]
00401233  |.^EB CD             \JMP SHORT 1f992c83.00401202
```

So, the virtual address range of the code performing the transformation is 00401202 (or 004011FB if we include initialization also) to 00401234.

This code executes with counter value 0 to 3D(61 in decimal). The counter is compared with 3D and the iteration continues if the value is less than or equal to. Otherwise, the code jumps out of the loop.

Below are the steps executed inside the loop:

- 1f992c83.00402008 address is stored into the EAX register (00402008 is an address in the data section and next 62 bytes will be modified starting from this address).
- Adds the counter value (after storing it in EDX) to this address in EAX and moves a byte from the resulting address in the data section to lower 8 bits in EAX.
- This byte is stored on the stack and XORed with '7F'.
- Again 1f992c83.00402008 is moved to the EAX register and counter value to EDX.
- The byte(result of XOR) is moved to lower 8 bits of ECX which is then moved to the address in the data section found by adding EAX and EDX.
- The counter value is incremented by 1.

So, in summary, each byte (62 bytes starting from address 00402008) is taken from the data section, XORed with '7F' and finally stored back at the same location from where it was fetched.

As we saw above, the XOR is used as the transformation technique. It is a very common method used for obfuscation purposes as it is easy and fast to implement.

The bytes in the data section before transformation:

```
00402000  01 00 00 00 98 3D 04 00 94 55 4C BF 7A 1A 1D 7F  ...˜=.”UL¿z
00402010  7F 97 DF 8E 80 80 7F 7F 7F C7 74 7F 7F 7F F6 8C  —ßŽ€€ÇtöŒ
00402020  F2 31 77 F2 29 73 B2 FF C7 7E 7F 7F 7F C4 7F 7F  ò1wò)s²ÿÇ~Ä
00402030  7F 7F B2 FF 97 AE 80 80 80 50 1D 16 11 50 0C 17  ²ÿ—®€€€PP.
00402040  7F F6 93 22 BC 00 00 00 FF FF FF FF 00 00 00 00  ö“"¼...ÿÿÿÿ....
00402050  5C 13 40 00 01 00 00 00 00 00 00 00 00 00 00 00  \@……….
```

The bytes in the data section after transformation:

```
00402000  01 00 00 00 98 3D 04 00 EB 2A 33 C0 05 65 62 00  ...˜=.ë*3Àeb.
00402010  00 E8 A0 F1 FF FF 00 00 00 B8 0B 00 00 00 89 F3  .è ñÿÿ...¸...‰ó
00402020  8D 4E 08 8D 56 0C CD 80 B8 01 00 00 00 BB 00 00  NV.Í€¸...»..
00402030  00 00 CD 80 E8 D1 FF FF FF 2F 62 69 6E 2F 73 68  ..Í€èÑÿÿÿ/bin/sh
00402040  00 89 EC 5D C3 7F 00 00 FF FF FF FF 00 00 00 00  .‰ì]Ã..ÿÿÿÿ....
00402050  5C 13 40 00 01 00 00 00 00 00 00 00 00 00 00 00  \@.....
```

As we can see above, the bytes from 00402008 to 00402045 (total 62 bytes) have been modified by the transformation code. There are many bytes equal to '7F' before transformation. It shows how the key gets reflected if we use XOR for obfuscation as all the 00 values get replaced with '7F'.

2. List the virtual address and type of instruction that transfers control to the dynamically generated code. Is there anything notable or unexpected about the mechanism used to transfer control to the dynamically generated code? Explain.

```
00401235  |> 8D45 FC       LEA EAX,DWORD PTR SS:[EBP-4]
00401238  |. 8D50 08       LEA EDX,DWORD PTR DS:[EAX+8]
0040123B  |. 8955 FC       MOV DWORD PTR SS:[EBP-4],EDX
0040123E  |. 8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
00401241  |. C700 08204000  MOV DWORD PTR DS:[EAX],1f992c83.00402008
00401247  |. C9            LEAVE
00401248  \. C3            RETN
```

The return instruction (RETN - C3) at address 00401248 transfers the control to the dynamically generated code. Return is generally used to transfer the control to the caller function so that the caller can continue execution starting from the address pushed on the stack at the time of the call.

The way it is used here is unexpected. The actual return address pushed on the stack was:

```
0240FF74  |004011B1  RETURN to 1f992c83.004011B1 from 1f992c83.004011F0
```

But, this address gets changed to *1f992c83.00402008* which we saw was the start address of the dynamically generated code. To perform this, it gets the address of local variable on the stack in EAX which is (EBP-4), adds 8 to this address (which gives the address in stack where return address it stored) and accesses it via DS storing the result in EDX (EDX becomes 0240FF74). This EDX value is stored on the stack at (EBP-4) from where it is moved to EAX. Finally, *1f992c83.00402008* is moved to the address present in EAX.

Therefore, instead of transferring the control to 1f992c83.004011B1, return instruction transfers the control to *1f992c83.00402008*. In this way, processor is forced to execute those dynamically generated instructions by manipulating the stack and changing the return address.

3. Excluding any initial control transfer instructions (e.g., jmp, call), list the reachably executable virtual address range of the dynamically generated code. What does the code do?

Below is the dynamically generated code ranging from address 00402008 to 00402045:

```
00402008   EB 2A            JMP SHORT 1f992c83.00402034 // Step 1 - JMP to 00402034
0040200A   33C0             XOR EAX,EAX // Step 3
0040200C   05 65620000      ADD EAX,6265 //step 4
00402011   E8 A0F1FFFF      CALL 1f992c83.004011B6  // step 5
00402016   0000             ADD BYTE PTR DS:[EAX],AL
00402018   00B8 0B000000    ADD BYTE PTR DS:[EAX+B],BH
0040201E   89F3             MOV EBX,ESI
00402020   8D4E 08          LEA ECX,DWORD PTR DS:[ESI+8]
00402023   8D56 0C          LEA EDX,DWORD PTR DS:[ESI+C]
00402026   CD 80            INT 80
00402028   B8 01000000      MOV EAX,1
0040202D   BB 00000000      MOV EBX,0
00402032   CD 80            INT 80
00402034   E8 D1FFFFFF      CALL 1f992c83.0040200A   // Step 2
00402039   2F               DAS
0040203A   6269 6E          BOUND EBP,QWORD PTR DS:[ECX+6E]
0040203D   2F               DAS
0040203E   73 68            JNB SHORT 1f992c83.004020A8
00402040   0089 EC5DC37F     ADD BYTE PTR DS:[ECX+7FC35DEC],CL
```

Return instruction at 00401248 transfers the control to instruction at 00402008. This is a JMP statement which takes us to address 00402034 which has a CALL instruction taking us to 0040200A. So, after these transfer instructions, the dynamically generated code which is reachably executable lies in the byte range (0040200A, 00402015). The other dynamically generated instructions can't be executed as we don't have a way to transfer the control and execute these instructions.

The reachably executable instructions just set EAX to 6265 (by using XOR and ADD instructions) and calls the instruction at 004011B6.

This call takes us to following portion of the code:

```
004011B6  |. E8 4D010000      CALL <JMP.&crtdll._cexit>
004011BB  |. 83C4 F4          ADD ESP,-0C
004011BE  |. 53               PUSH EBX                              ; /ExitCode
004011BF  \. E8 84010000      CALL <JMP.&KERNEL32.ExitProcess> ; \ExitProcess
```

These instructions just call crtdll._cexit(), decrements ESP by 0C and calls kernel32.ExitProcess() with ExitCode 7FFDE000 pushed on the stack.

4. Describe what you believe are the intentions of the program. Is it malicious? How does your assessment compare with the classification results of major antivirus products?

The program is a benign one, probably developed just to help us learn the binary analysis technique. It doesn't perform any malicious activity. The dynamically generated code simply terminates the process after executing some benign instructions. So, it is not an malicious program.

This assessment totally differs from the classification results of major antivirus products. 39 out of the 55 antivirus products provided on virustotal flag the PE file as malicious (many report it as trojan). This may be because of the behavior of the program as it uses obfuscation technique and tries to generate some code dynamically by writing in the data section. But, it doesn't prove that the program has some malicious intent. So, this shows that the detection by AV products need not be because of some actual malicious behavior. It is based on some suspicious activities (which are commonly used by malware) performed by the process even though it is not malicious.