Announcing the New Null Byte IRC Channel
Recognition of Excellence at Null Byte (Fellows, Awards, Recommendations, & Certifications)
How to Post to Null Byte [4.10.15 Revision]
The Dark Side of Kickstarter
Keeping Your Hacking Identity Secret: How to Become a Ghost Hacker #4
4 Ways to Crack Facebook Passw and How to Pro Yourself from T

An Extended Guide on Runtime Crypters

Posted By dontrustme 501 last month

11 KUDOS

Hello again, folks! I'm back with another (final) guide on runtime crypters which is an extension on my previous runtime crypter guide. If you have not read it yet, I highly recommend that you do since the fundamental theory of the PE format. Again, this will be a Windows-specific guide, so I'll repeat this: *If you're a hater and cringe at the slightest sound of Microsoft's grotesque baby, feel free to close this tab, delete your history, exit your browser, pour some oil onto your machine and sanitize it with fire! Else, please continue onwards.*

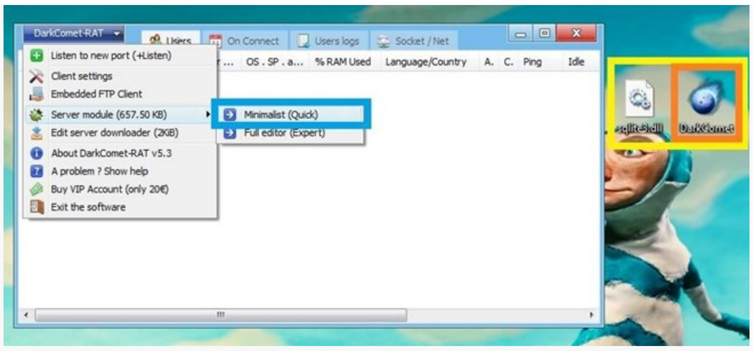
This **advanced** guide will be discussing a concept which has some relation to packing technology so if you're interested in that, this may serve as a really basic introduction. I currently do not know what this technique is called but don't worry, I will explain its method and procedure so if anyone knows what it's called, just put it down in the comment section below, or not, whatever you want... I will be putting up my source code if anyone wants to have a look at what's happening in the technical code viewpoint but since this is purely conceptual, please do not expect it to work on all applications and if you're using it to test malware, do not expect it to be undetectable. This means that I will not be running it under an active antivirus environment. (Sorry!).

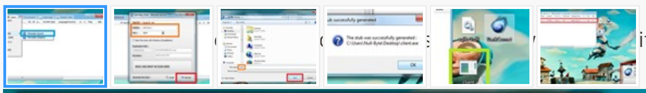
Note: I am still currently learning more about this so if I happen to get any information incorrect, please leave a comment below or drop a message in my inbox and I will try to patch it ASAP.

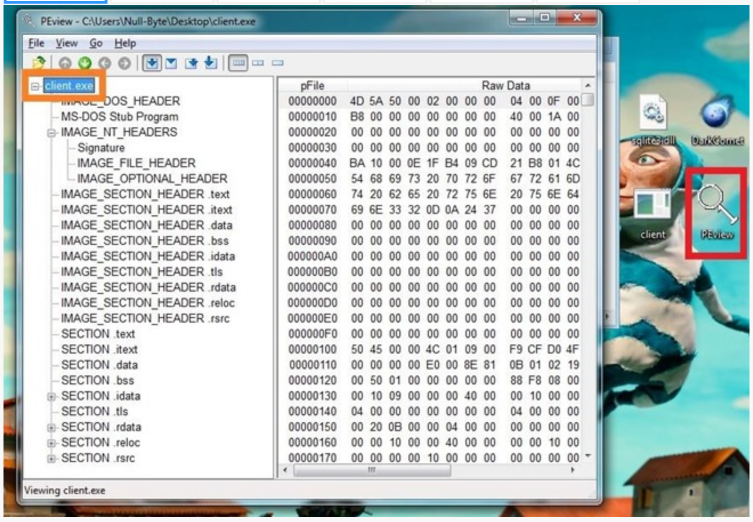
Disclaimer: This is an article which shows how the runtime crypter works. It is meant to be a guide, not a tutorial so not every little piece of information will be provided.

Visualizing the PE Format

First, we will be analyzing the target file for our proof of concept, i.e. we will be using a Dark Comet client executable. Let's create one using the Dark Comet RAT and then analyze it.




it.


PEview - C:\Users\Null-Byte\Desktop\client.exe

```
client.exe
    IMAGE_DOS_HEADER
    MS-DOS Stub Program
    IMAGE_NT_HEADERS
        Signature
        IMAGE_FILE_HEADER
        IMAGE_OPTIONAL_HEADER
    IMAGE_SECTION_HEADER .text
    IMAGE_SECTION_HEADER .itext
    IMAGE_SECTION_HEADER .data
    IMAGE_SECTION_HEADER .bss
    IMAGE_SECTION_HEADER .idata
    IMAGE_SECTION_HEADER .tls
    IMAGE_SECTION_HEADER .rdata
    IMAGE_SECTION_HEADER .reloc
    IMAGE_SECTION_HEADER .rsrc
    SECTION .text
    SECTION .itext
    SECTION .data
    SECTION .bss
    SECTION .idata
    SECTION .tls
    SECTION .rdata
    SECTION .reloc
    SECTION .rsrc
```

```
pFile     Raw Data
00000000  4D 5A 50 00 02 00 00 00  04 00 0F 00
00000010  B8 00 00 00 00 00 00 00  40 00 1A 00
00000020  00 00 00 00 00 00 00 00  00 00 00 00
00000030  00 00 00 00 00 00 00 00  00 00 00 00
00000040  BA 10 00 0E 1F B4 09 CD  21 B8 01 4C
00000050  54 68 69 73 20 70 72 6F  67 72 61 6D
00000060  74 20 62 65 20 72 75 6E  20 75 6E 64
00000070  69 6E 33 32 0D 0A 24 37  00 00 00 00
00000080  00 00 00 00 00 00 00 00  00 00 00 00
00000090  00 00 00 00 00 00 00 00  00 00 00 00
000000A0  00 00 00 00 00 00 00 00  00 00 00 00
000000B0  00 00 00 00 00 00 00 00  00 00 00 00
000000C0  00 00 00 00 00 00 00 00  00 00 00 00
000000D0  00 00 00 00 00 00 00 00  00 00 00 00
000000E0  00 00 00 00 00 00 00 00  00 00 00 00
000000F0  00 00 00 00 00 00 00 00  00 00 00 00
00000100  50 45 00 00 4C 01 09 00  F9 CF D0 4F
00000110  00 00 00 00 E0 00 8E 81  0B 01 02 19
00000120  00 50 01 00 00 00 00 00  88 F8 08 00
00000130  00 10 09 00 00 00 40 00  00 10 00 00
00000140  04 00 00 00 00 00 00 00  04 00 00 00
00000150  00 20 0B 00 00 04 00 00  00 00 00 00
00000160  00 00 10 00 00 40 00 00  00 00 10 00
00000170  00 00 00 00 10 00 00 00  00 00 00 00
```

Viewing client.exe

We can see all of the headers listed and each section as well. Great!

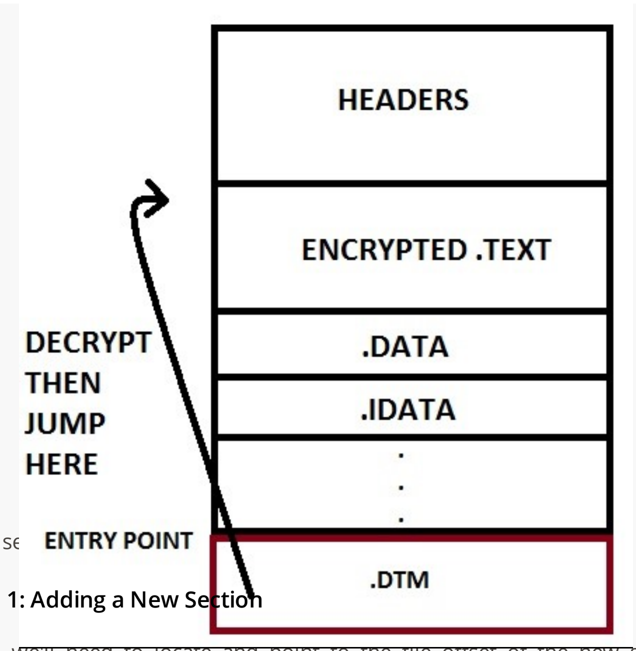
Note: The *.itext* section is another code section so for simplicity's sake, we will ignore it.

Creating a Concept

We obviously want the file to be obfuscated on disk and we can do this with some encryption method. That's easy. We also want it to decrypt the file on runtime like what our previous method did however, this time, we do not want to use a loader to write the target PE file into memory and then execute it... We want it to decrypt *itself* and then have execution jump to the starting point. How can we do this? Let's go more into detail...

... but just briefly, let me explain how code is executed. When a program is launched, its PE file is transferred into memory which we had quickly looked over in the previous guide. Once this happens, the *Instruction Pointer* (EIP for 32-bit programs, RIP for 64-bit programs) is pointed to the *AddressOfEntryPoint* member listed in the Optional header which is its value + the *ImageBase* member in memory, i.e. the execution will start at memory location 0x0048F888.

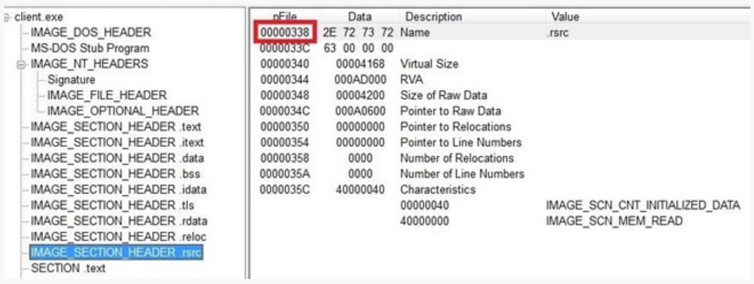
What we want to do is create a *new* section inside the target PE file, have a decryption routine written inside it and point the *AddressOfEntryPoint* to the start of our section's routine. We will also need to jump back to the original *AddressOfEntryPoint* when our routine has finished. So, overall, code execution will start at our code, have it decrypt itself and then jump back to the original entry point. Sounds simple enough?



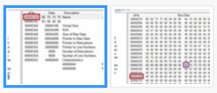
Let's se

Step 1: Adding a New Section

First, we'll need to locate and point to the file offset of the new section *header*, i.e. the last section header's offset + the size of a section header (0x28). Let's find it in PEview:



```
client.exe                          pFile     Data         Description               Value
  IMAGE_DOS_HEADER                  00000338  2E 72 73 72  Name                      .rsrc
  MS-DOS Stub Program               0000033C  63 00 00 00
  IMAGE_NT_HEADERS                  00000340  00004168     Virtual Size
    Signature                       00000344  000AD000     RVA
    IMAGE_FILE_HEADER               00000348  00004200     Size of Raw Data
    IMAGE_OPTIONAL_HEADER           0000034C  000A0600     Pointer to Raw Data
  IMAGE_SECTION_HEADER .text        00000350  00000000     Pointer to Relocations
  IMAGE_SECTION_HEADER .itext       00000354  00000000     Pointer to Line Numbers
  IMAGE_SECTION_HEADER .data        00000358  0000         Number of Relocations
  IMAGE_SECTION_HEADER .bss         0000035A  0000         Number of Line Numbers
  IMAGE_SECTION_HEADER .idata       0000035C  40000040     Characteristics
  IMAGE_SECTION_HEADER .tls                                00000040                  IMAGE_SCN_CNT_INITIALIZED_DATA
  IMAGE_SECTION_HEADER .rdata                              40000000                  IMAGE_SCN_MEM_READ
  IMAGE_SECTION_HEADER .reloc
  IMAGE_SECTION_HEADER .rsrc
  SECTION .text
```



The last section, *.rsrc* is located at file offset 0x338 so according to our calculation, 0x338 + 0x28 = 0x360, therefore our new section's header should start at file offset 0x360. From here, we just need to add in the appropriate information such as the *Name*, *VirtualSize*, *VirtualAddress* (RVA), *SizeOfRawData*, *PointerToRawData* and *Characteristics*.

- Our *VirtualSize* member tells how much information is to be allocated in memory
- The *VirtualAddress* member is the relative offset of the section's address in memory
- The *SizeOfRawData* determines the size of the section's information on disk
- The *PointerToRawData* tells us the file offset of the beginning of our section.
- Finally, the *Characteristics* tells us how memory should interpret and handle the code. For this, we require it to at least contain the code and execute flags. We can just copy the same flags from the original code section.

Now we need to add in the section itself. Of course, there's nothing special to doing this, all we need to do is append our routine to the file. The section header will automagically register it.

All good!

Step 2: Obfuscating the Code Section

Since we (obviously) want the information to be obfuscated, we need to write up a function to this. We can go further and obfuscate the all of the sections but, again, for simplicity's sake, we will only modify the code. This is a pretty trivial task. We need to locate the beginning of the code section using its header's *PointerToRawData* and then obfuscate the length of *SizeOfRawData*. I have decided to use the same XOR encryption method and key as I did in my scantime crypter.

A cinch. Next!

Step 3: Integrating Our Routine

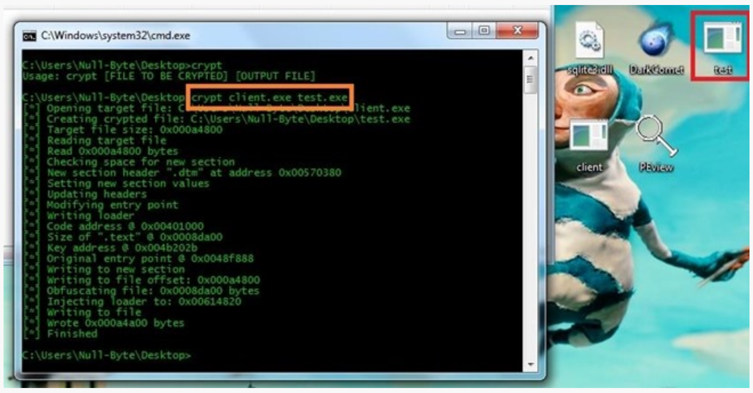
As I've stated before, we need to write some sort of decryption routine to decrypt its original code section on runtime and then jump back to the original entry point (OEP). How do we do this? We can define an assembler routine to take the code section's virtual offset, *VirtualAddress* (RVA), and decrypt it for the code section's *SizeOfRawData* to reverse our previous step. Once that's finished, we can *push* the OEP address and then call *ret*.

To add this onto the file, just append it as I've mentioned before.

Easy? Maybe...?

Analyzing Our Modified File

First, let's build our modified file...



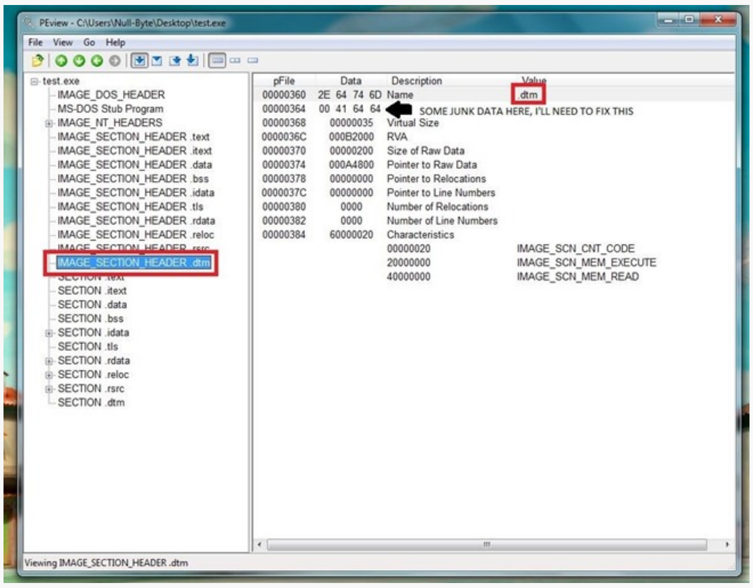
```
C:\Users\Null-Byte\Desktop>crypt
Usage: crypt [FILE TO BE CRYPTED] [OUTPUT FILE]

C:\Users\Null-Byte\Desktop>crypt client.exe test.exe
[*] Opening target file: C:\Users\Null-Byte\Desktop\client.exe
[*] Creating crypted file: C:\Users\Null-Byte\Desktop\test.exe
[*] Target file size: 0x000a4800
[*] Reading target file
[*] Read 0x000a4800 bytes
[*] Checking space for new section
[*] New section header ".dtm" at address 0x00570380
[*] Setting new section values
[*] Updating headers
[*] Modifying entry point
[*] Writing loader
[*] Code address @ 0x00401000
[*] Size of ".text" @ 0x0008da00
[*] Key address @ 0x004b202b
[*] Original entry point @ 0x0048f888
[*] Writing to new section
[*] Writing to file offset: 0x000a4800
[*] Obfuscating file: 0x0008da00 bytes
[*] Injecting loader to: 0x00614820
[*] Writing to file
[*] Wrote 0x000a4a00 bytes
[*] Finished

C:\Users\Null-Byte\Desktop>
```

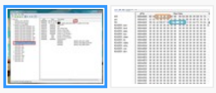
(Don't forget the green color to make ourselves look 1337!)

And here it is in PEview:


PEview - C:\Users\Null-Byte\Desktop\test.exe
File View Go Help
SOME JUNK DATA HERE, I'LL NEED TO FIX THIS
Viewing IMAGE_SECTION_HEADER .dtm

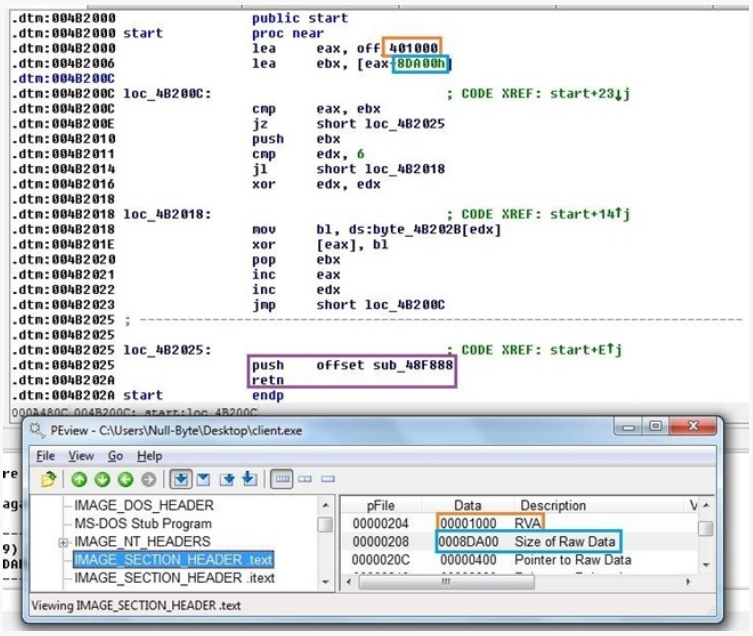
```
test.exe
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .itext
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .bss
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .tls
  IMAGE_SECTION_HEADER .rdata
  IMAGE_SECTION_HEADER .reloc
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .dtm
  SECTION .text
  SECTION .itext
  SECTION .data
  SECTION .bss
  SECTION .idata
  SECTION .tls
  SECTION .rdata
  SECTION .reloc
  SECTION .rsrc
  SECTION .dtm
```

pFile	Data	Description	Value
00000360	2E 64 74 6D	Name	.dtm
00000364	00 41 64 64		
00000368	00000035	Virtual Size	
0000036C	000B2000	RVA	
00000370	00000200	Size of Raw Data	
00000374	000A4800	Pointer to Raw Data	
00000378	00000000	Pointer to Relocations	
0000037C	00000000	Pointer to Line Numbers	
00000380	0000	Number of Relocations	
00000382	0000	Number of Line Numbers	
00000384	60000020	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ



In the orange, we see the virtual address of the *.text* section and in the blue is the virtual address of the OEP. If it looks a bit backwards to you, don't forget to apply *endianness*!

A little extra static and dynamic analysis on what's happening using IDA Pro.



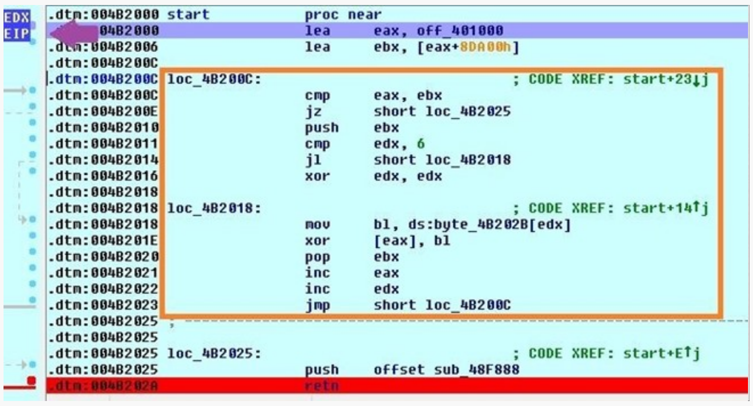
```
.dtm:004B2000                  public start
.dtm:004B2000 start            proc near
.dtm:004B2000                  lea     eax, off 401000
.dtm:004B2006                  lea     ebx, [eax+8DA00h]
.dtm:004B200C
.dtm:004B200C loc_4B200C:                            ; CODE XREF: start+23↓j
.dtm:004B200C                  cmp     eax, ebx
.dtm:004B200E                  jz      short loc_4B2025
.dtm:004B2010                  push    ebx
.dtm:004B2011                  cmp     edx, 6
.dtm:004B2014                  jl      short loc_4B2018
.dtm:004B2016                  xor     edx, edx
.dtm:004B2018
.dtm:004B2018 loc_4B2018:                            ; CODE XREF: start+14↑j
.dtm:004B2018                  mov     bl, ds:byte_4B202B[edx]
.dtm:004B201E                  xor     [eax], bl
.dtm:004B2020                  pop     ebx
.dtm:004B2021                  inc     eax
.dtm:004B2022                  inc     edx
.dtm:004B2023                  jmp     short loc_4B200C
.dtm:004B2025 ; ---------------------------------------------------------------
.dtm:004B2025
.dtm:004B2025 loc_4B2025:                            ; CODE XREF: start+E↑j
.dtm:004B2025                  push    offset sub_48F888
.dtm:004B202A                  retn
.dtm:004B202A start            endp
000A480C 004B200C: start:loc_4B200C
```

PEview - C:\Users\Null-Byte\Desktop\client.exe

pFile	Data	Description
00000204	00001000	RVA
00000208	0008DA00	Size of Raw Data
0000020C	00000400	Pointer to Raw Data

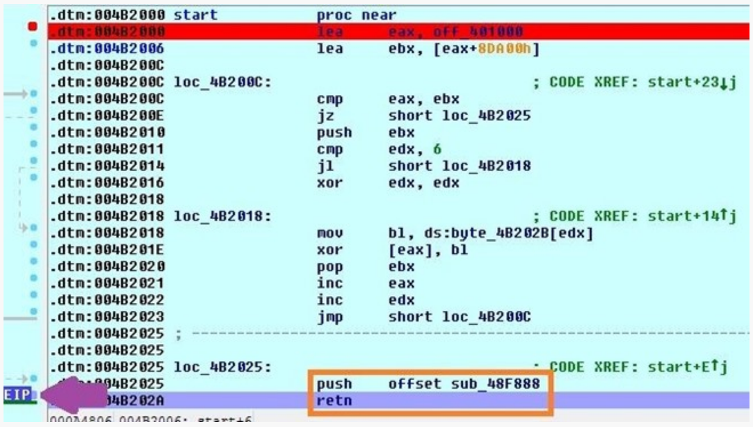
Viewing IMAGE_SECTION_HEADER .text

Here is the disassembly of the assembler routine. Notice in the orange boxes is the *.text* section's virtual address and in the blue boxes is the size of *.text* section's data. In the purple are the instructions to jump back to the offset of the OEP.



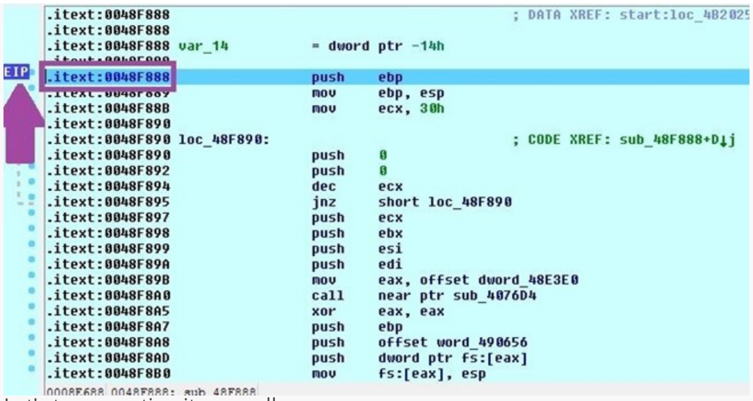
```
.dtm:004B2000 start            proc near
.dtm:004B2000                  lea     eax, off_401000
.dtm:004B2006                  lea     ebx, [eax+8DA00h]
.dtm:004B200C
.dtm:004B200C loc_4B200C:                            ; CODE XREF: start+23↓j
.dtm:004B200C                  cmp     eax, ebx
.dtm:004B200E                  jz      short loc_4B2025
.dtm:004B2010                  push    ebx
.dtm:004B2011                  cmp     edx, 6
.dtm:004B2014                  jl      short loc_4B2018
.dtm:004B2016                  xor     edx, edx
.dtm:004B2018
.dtm:004B2018 loc_4B2018:                            ; CODE XREF: start+14↑j
.dtm:004B2018                  mov     bl, ds:byte_4B202B[edx]
.dtm:004B201E                  xor     [eax], bl
.dtm:004B2020                  pop     ebx
.dtm:004B2021                  inc     eax
.dtm:004B2022                  inc     edx
.dtm:004B2023                  jmp     short loc_4B200C
.dtm:004B2025 ;
.dtm:004B2025
.dtm:004B2025 loc_4B2025:                            ; CODE XREF: start+E↑j
.dtm:004B2025                  push    offset sub_48F888
.dtm:004B202A                  retn
```

I have set a breakpoint at the beginning of code execution - remember, I modified the entry point to the address of the new section. In the large orange box is the decryption routine. EIP will run through this and decrypt everything that we have encrypted on disk in runtime.



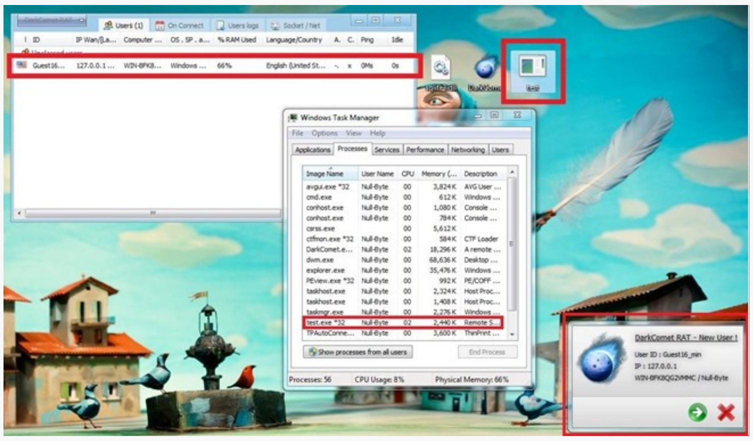
```
.dtm:004B2000 start            proc near
.dtm:004B2000                  lea     eax, off_401000
.dtm:004B2006                  lea     ebx, [eax+8DA00h]
.dtm:004B200C
.dtm:004B200C loc_4B200C:                            ; CODE XREF: start+23↓j
.dtm:004B200C                  cmp     eax, ebx
.dtm:004B200E                  jz      short loc_4B2025
.dtm:004B2010                  push    ebx
.dtm:004B2011                  cmp     edx, 6
.dtm:004B2014                  jl      short loc_4B2018
.dtm:004B2016                  xor     edx, edx
.dtm:004B2018
.dtm:004B2018 loc_4B2018:                            ; CODE XREF: start+14↑j
.dtm:004B2018                  mov     bl, ds:byte_4B202B[edx]
.dtm:004B201E                  xor     [eax], bl
.dtm:004B2020                  pop     ebx
.dtm:004B2021                  inc     eax
.dtm:004B2022                  inc     edx
.dtm:004B2023                  jmp     short loc_4B200C
.dtm:004B2025 ; ---------------------------------------------------------------
.dtm:004B2025
.dtm:004B2025 loc_4B2025:                            ; CODE XREF: start+E↑j
.dtm:004B2025                  push    offset sub_48F888
.dtm:004B202A                  retn
000A4806 004B2006: start+6
```

EIP is now at the end of our assembler routine and is about to return back to the OEP...



```
.itext:0048F888                                         ; DATA XREF: start:loc_482025
.itext:0048F888
.itext:0048F888 var_14          = dword ptr -14h
.itext:0048F888
.itext:0048F888                 push    ebp
.itext:0048F889                 mov     ebp, esp
.itext:0048F88B                 mov     ecx, 30h
.itext:0048F890
.itext:0048F890 loc_48F890:                             ; CODE XREF: sub_48F888+D↓j
.itext:0048F890                 push    0
.itext:0048F892                 push    0
.itext:0048F894                 dec     ecx
.itext:0048F895                 jnz     short loc_48F890
.itext:0048F897                 push    ecx
.itext:0048F898                 push    ebx
.itext:0048F899                 push    esi
.itext:0048F89A                 push    edi
.itext:0048F89B                 mov     eax, offset dword_48E3E0
.itext:0048F8A0                 call    near ptr sub_4076D4
.itext:0048F8A5                 xor     eax, eax
.itext:0048F8A7                 push    ebp
.itext:0048F8A8                 push    offset word_490656
.itext:0048F8AD                 push    dword ptr fs:[eax]
.itext:0048F8B0                 mov     fs:[eax], esp
0008F688 0048F888: sub_48F888
```

Let's try executing it normally...



Yep! Everything is working perfectly fine! Awesome!

Conclusion

Again, if you're confused, please do some more research - there's more than enough resources online for this. If you have not read my previous guide on runtime crypters, please do so since the theory of the PE format is explained (to some extent anyway) over there.

Hope you've all enjoyed this two-part series and have familiarized yourselves with different, very creative and magical PE manipulation! Thanks to those who have morally supported me during the development of this program! (I'm looking at you, dill_.) I'm not sure about what I'll write about next time, but I sure hope that it'll be as informative and interesting!

That's all folks!

dtm.

P.S. Oh, right, you're looking for the source code, yes? Well, I've delayed you long enough, just remember that it is purely proof-of-concept and will not work for all applications and certainly will not guarantee undetectable malware... Oh, and about comments, they're not too detailed as of writing this but I will certainly fix this issue! Follow me over to Github!

See Also

- A Guide on Runtime Crypters
- Security-Oriented C Tutorial 0xFB - A Simple Crypter
- Reversing and Analyzing a Runtime Crypter

Show More...

Join the Discussion

Subscribe OFF

1

PICOFLA MINGO

Congratulations dtm!. Very good work

Now I know how it works on Windows. But, just for my education, is it not needed to fix memory permissions on Windows at run-time?

last month Reply

DONTRUSTME

1

Thanks, m8. What do you mean by "fix memory permissions"?

last month Reply

PICOFLAMINGO

2

Sorry for the vague formulation of the question. Let me elaborate it.

On Linux (with ELF binaries), the section permissions are honoured. The **.text** section usually does not have **WRITE** permissions. If you try to write in the **.text** section your program will just segfault. This is managed by the MMU (Memory Management Unit) inside the Processor at the HW level (AFAIK). It is actually the mechanism used to implement virtual memory.

So, before writing (decrypting) to the **.text** section you need to add the write permission to that section (usually a call to **mprotect**). I checked your asm and it seems to just do the XORing of the whole section (no permission changes).

My question is: Does Windows prevents writing in the .text segment by default as it happens on Linux?. Just to know, I haven't done anything on windows in years, and I'm just curious.

Note: I think you can make your key a multiple of the page alignment and do the XOR on 16/32bits (on bx, or ebx) at once (depending on the alignment), instead of byte by byte. I do not think there will be a big improvement on performance, but your loader will probably be shorter. You will also be able to just bitwise AND a mask to iterate through the key.

last month Reply

DONTRUSTME

2

Yep, you do need to set the Characteristics to include a write, see here: crypt.c, line 186.

As for alignment and XORing in 16-/32-bit blocks(?), sure, that's totally possible. I never thought of doing that so thanks for mentioning it. I was never intending on doing anything more than a basic byte-by-byte because I wanted to keep it real simple.

last month Reply

PICOFLAMINGO

1

OK, I see. You change the permissions in the file itself.

Thanks!

last month Reply

_PAROXYSM

1

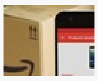
Hi! I've been working on my crypter for executable files, and I'm just wondering how easy it would be to get the algorithm and the key I used in the stub, and how can I protect my stub. I'm not an expert in reverse engineering

yesterday Reply

Share Your Thoughts

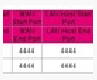
The Dark Side of Kickstarter

4 Ways to Crack a Facebook Password and How to Protect Yourself from Them


How to Always Get the Best Deal on Amazon

5 Clever Ways to Put Your Expired Spices to Work

15 Mind-Blowing Things You Can Do with Leftover Pickle Juice


WAN Meterpreter Session Using Msfvenom Not Working?

Arts

Arts & Crafts
Beauty & Style
Dance
Fine Art
Music & Instruments

Science & Tech

Autos, Motorcycles & Planes
Computers & Programming
Disaster Preparation
Education
Electronics
Film & Theater
Software
Weapons

Lifestyle

Alcohol
Business & Money
Dating & Relationships
Diet & Health
Family
Fitness
Food
Home & Garden
Hosting & Entertaining
Language
Motivation & Self Help
Outdoor Recreation
Pets & Animals
Pranks & Cons
Spirituality
Sports
Travel

Gaming

Gambling
Games
Hobbies & Toys
Magic & Parlor Tricks
Video Games