Modifying Metasploit Shellcode Decoders to Bypass Static Analysis

Alexander Bode abode@os3.nl
Jelle Ermerins jermerins@os3.nl

May 31, 2020

Abstract—In this research, the Shikata Ga Nai decoder is analysed to find a method that allows decreasing detection of shellcode decoders during static analysis. By analysing the decoder stub of a payload encoded with the Shikata Ga Nai encoder and looking at current static detection methods that detect the Shikata Ga Nai decoder stub, four critical parts of the decoder stub are identified, of which any can be modified to bypass static detection. Using a hex editor and a custom created tool, instructions of the Shikata Ga Nai decoder stub are modified to bypass static detection. Techniques that are used to bypass static detection are polymorphism and alternative instruction encoding. The payloads with the modified decoder stubs are tested against four different static detection solutions. The modified payloads successfully bypassed all static detection systems that were tested during our experiments. This research shows that using our custom tool and a hex editor, while applying the techniques polymorphism and alternative instruction encoding, provide a robust method for modifying any x86 shellcode to bypass static detection. Detection rules could be improved by including the modifications in this research. However, our proposed method demonstrates the ease of the process being provided with the right tools. As far as static detection concerned, this results in a never-ending cat and mouse game.

I. INTRODUCTION

The goal of penetration testing is to identify weaknesses in an organization’s security. Both attackers and penetration testers may apply various techniques to enumerate and exploit a system, often while remaining stealthy. An increasingly popular tool for developing and running exploit code against target systems is the Metasploit Framework. The Metasploit Framework includes a variety of features, including but not limited to configuring remote code execution exploits, backdooring executables and automated encoding of payloads. The popularity of the tool has prompted the providers of anti-virus and intrusion detection or prevention systems to come up with methods for detecting payloads that are generated by Metasploit.

Avoiding detection might be of importance during the exploitation phase of penetration testing. This could be achieved by encoding payloads using an encoder. The primary detection techniques of the majority of security vendors include static code analysis and behavioural analysis. Although payloads can be encoded to avoid detection, current static detection methods can detect encoded payloads by identifying the decoder part of the payload [1] [2]. In this study, we will present an approach for modifying Metasploit’s shellcode decoders to decrease detection during static analysis by anti-virus and intrusion detection systems. An analysis of Metasploit’s Shikata Ga Nai decoder will be presented and discussed in order to show ways to decrease or bypass static detection.

To avoid detection using existing encoders, security testers need to uniquely customize the decoder of the payload, while maintaining the payload’s functionality. To determine how these decoders could be altered to avoid detection, we came up with the following research question:

Which method allows decreasing detection of shellcode decoders during static analysis?

To answer our research question, we have defined the following sub-questions:

• What techniques are used in the Shikata Ga Nai decoder to bypass static detection?
• How are shellcode decoders detected by static analysis?
• How can shellcode decoders be modified to bypass static analysis?

II. RELATED WORK

This section gives an overview of work that has previously been conducted on Shikata Ga Nai and its detection.

In a blog post by Steve Miller et. al., the algorithm of the Shikata Ga Nai encoder has been analysed [3]. This is done by creating a plain Shikata Ga Nai encoded payload and looking at the binary in a disassembler. Furthermore, they present a YARA rule that detects a Shikata Ga Nai encoded payload by matching the behaviour of the Shikata Ga Nai algorithm. They describe that static detection could be easily defeated by including evasion techniques like using different registers and adding junk code.
Furthermore, in a blog post by Nick Hoffman et. al., the possible instructions that can be used for Shikata Ga Nai are discussed [1]. They generated a sample set of 236 permutations of a Shikata Ga Nai encoded payload. To detect these payloads, they wrote a YARA rule. With this rule, they were able to detect all of the 236 payloads. The research shows that it is still possible to statically detect Shikata Ga Nai payloads, because some parts of the Shikata Ga Nai algorithm consists of hardcoded instructions or a finite set of possible instructions. At the end of the article, some suggestions are proposed to reduce the chance of being detected using static detection. One of these suggestions is to support more options to retrieve the location of the instruction pointer, which we will use to modify the Shikata Ga Nai decoder, as can be seen in Section V.

Another method that can be implemented to bypass static detection is the use of encoded instructions, which are the same but with a different encoding. This method is explained in a DEFCON presentation by XlogicX [3]. We will also implement this technique while modifying the Shikata Ga Nai decoder.

Finally, in a blog post by Marcos Valle, the Shikata Ga Nai encoder is described in-depth, where each step of the Shikata Ga Nai algorithm is described with the corresponding assembly code [4].

The previous work shows that static detection can be used to detect Shikata Ga Nai encoded payloads, however, previous work also shows that Shikata Ga Nai be can be modified to evade static detection. In our research, we will modify the Shikata Ga Nai decoder so that static detection can be evaded. In our research, the proposed methods from Nick Hoffman et. al [1] and the encoded instructions method from XlogicX [3] will be put into practice for modifying the Shikata Ga Nai decoder. The pattern matching rule that was written by Steve Miller et. al. will be one of the static detection methods we use to test the evasion of static detection [2].

### III. BACKGROUND

In this section, we give a general overview of encoded shellcode and the Shikata Ga Nai decoder. We continue by describing the techniques that are used by the decoder that decodes the encoded shellcode. Each mentioned technique is taken into consideration, as it may offer us the possibility to reduce or bypass detection.

#### A. Encoded Shellcode

Encoded shellcode consists of two parts: the decoder stub and the encoded payload. The instructions in the decoder stub are responsible for decoding the encoded payload upon execution of the shellcode. The decoder stub consists of instructions that load the key, obtain the value of the instruction pointer and a decoder loop that decodes the encoded payload. The terms Shikata Ga Nai decoder stub, Shikata Ga Nai decoder and decoder stub are used interchangeably in this report.

#### B. The Shikata Ga Nai Decoder

The Shikata Ga Nai decoder is a polymorphic XOR additive feedback decoder [5]. The encoded payload is decoded by the Shikata Ga Nai decoder using iterations of bitwise XOR instructions. The Shikata Ga Nai decoder decodes the encoded shellcode instructions by performing bitwise XOR binary operations on these instructions with a key that is randomly chosen by the Shikata Ga Nai encoder. On every iteration of the XOR loop, the key is altered by adding the result of the instruction after the XOR operation to the key itself. Hence the name XOR additive feedback decoder.

Decoder stubs of Shikata Ga Nai shellcode containing identical payloads are likely not to be the same. This is because the generator of the Shikata Ga Nai encoder uses a permutation of instructions and registers, which provide the same functionality across different the generated decoders [6]. For example, random registers are used to store the decoding key. This makes the Shikata Ga Nai encoder a polymorphic encoder, where different permutations of instructions and registers are used while generating encoded shellcode.

The execution of Shikata Ga Nai’s decoder stub consists of four steps:

1) Store a random key
2) Retrieve the value of the instruction pointer with floating-point unit instructions
3) XOR decode 4 subsequent bytes of the payload
4) Update the decoding key

Figure [I] in Appendix [A] shows a control-flow graph of a payload encoded with the Shikata Ga Nai encoder. The first instruction stores the key 0x36def162 in the EAX register. The decoder stub needs to know where the encoded payload starts to perform bitwise XOR operations on the encoded instructions using the decoding key. This is done by obtaining the current value of the instruction pointer and is necessary to calculate the offset of the encoded payload. Shikata Ga Nai uses floating-point unit instructions to obtain the instruction pointer’s value. The first step is to use a floating-point unit instruction that pre-populates the FPUDataPointer, which contains the address of the previously executed floating-point unit instruction. In the example in Figure [I] the fcmovu st(0),st(3) instruction is used. Next, the value of the instruction pointer is obtained by executing the fstenv [esp-0xc] instruction. This instruction pushes the FPUDataPointer, located at an offset of 0x0c or 12 bytes in the floating-point unit environment, on top of the stack [I]. The address of the previous floating-point unit instruction, relative to the value of the instruction pointer, gets popped off the stack and stored in the EDX register with the (POP EDX) instruction.

The decoding XOR loop consists of four instructions, located at addresses 0x00417010 up to 0x00417019, as shown in Figure [I]. On each iteration, four bytes are
added to the register that stored the value of the instruction pointer in this case the EDX register. Furthermore, bitwise XOR operations are performed on the instruction located at [EDX+0xf] with the key stored in EAX. This address is the first encoded instruction, relative to the instruction pointer with an offset of 0xf. Finally, the key gets updated by adding the result of the bitwise XOR operation on the instruction to the decoding key, which is stored in the EAX register.

The number of iterations in the decoder loop is determined by the CL register. Before the decoder loop starts, a MOV CL, 0xb instruction loads 0xb into the CL register. This is the length of the encoded payload. After a XOR instruction in the decoder loop, the LOOP instruction decrements the CL register by one. When the CL register is equal to zero, all instructions of the encoded payload are decoded and the loop is finished, after which the decoded payload is executed.

C. Encoding Techniques

The Shikata Ga Nai encoder that generates the decoder stub implements various techniques to evade static detection. These techniques result in many different variations of the decoder stub. This makes it harder to statically detect the decoder stub of the Shikata Ga Nai payload.

1) Instruction Substitution: One of the techniques implemented in Shikata Ga Nai is instruction substitution. This means that multiple instructions can be used to provide the same result in the decoder stub. The floating-point unit instructions, which retrieve the value of the instruction pointer in Shikata Ga Nai, are an example of dynamic instruction substitution. The control-flow graph in Figure 1 shows that the instruction fcmovu st(0),st(3) is used to populate the FPUDataPointer. However, multiple instructions can be used to do this. The fpu_instructions function in the source code of Shikata Ga Nai returns a set of floating-point unit instructions that can be used for populating the FPUDataPointer.

2) Reordering of Instructions: Looking at multiple permutations of the decoder stub in Figure 1 and 2 in Appendix A we see that the order of the three instructions in the decoder loop can differ. Furthermore, the order of the first three instructions that store the key and initialize the instruction pointer may also differ.

3) Random Registers: The registers where the loop counter, the value of the instruction pointer and the decoding key that are stored may also differ. Figure 1 shows that the key is stored in the EAX register, the instruction pointer is stored in the EDX register and the counter in the ECX register. Looking at Figure 2 also in Appendix A the key is now stored in the EDX register and the instruction pointer is stored in the EAX register.

D. Static Detection

Static detection of encoded payloads can be done by recognizing the behaviour of the decoder stub, e.g., by matching binary patterns using YARA rules. A limited number of permutations exist within several parts of the Shikata Ga Nai decoder stub and for that reason, a complex YARA rule can be built to match these permutations. Take for example the first instructions in the decoder stub shown in Figure 1 and 2 in Appendix A. We know that there is a limited set of floating-point unit instructions that pre-populate the FPUDataPointer. We also know that the instruction that pushes the instruction pointer on the stack is always the same (fstenv [esp-0xc]) and only occurs after the other floating-point unit instruction. Furthermore, the instruction pointer will always be popped off the stack. The only variation is the register which holds the value of the instruction pointer. This also is the case for zeroing out the CL register, which used as a counter for the XOR loops, as well as loading the value of the counter into the CL register.

There is a limit to the number of possible permutations within the decoder loop. In the first three instructions of the loop, a random register, one that holds the value of the instruction pointer, gets incremented by four bytes. A bitwise XOR operation will be performed on a set of 4 bytes of the payload, after which the decoding key gets updated. Note that the key can only be updated after a bitwise XOR operation has been performed on the instruction, which reduces the number of permutations. The fourth instruction in the loop is always the same LOOP –9 instruction with an opcode of E2 F5.

With this information on the decoder stub and the limited permutations and registers that are used, a YARA rule can be created that can detect the variations of the decoder stub. An example of a YARA rule to detect the Shikata Ga Nai decoder stub can be seen in Listing 2 in Appendix A, which is written by Steven Miller of FireEye. Listing 1 shows a part of the YARA rule, which is shown in Appendix A, that can identify a set of permutations of the decoder stub. The first part of the rule, i.e., B8 ?? ?? ?? ??), is the part where the decoding key gets stored into a random register. In this example, the key gets stored in the EAX register, which corresponds to the B8 opcode and has the x86 mnemonic MOV EAX. The next four bytes are the decoding key itself and is chosen randomly. The next part is the floating-point unit instruction to retrieve the value of the instruction pointer, which is always the same, i.e., (D9 74 24 F4). Afterwards, the instruction pointer gets stored in a random register. This is done using a POP instruction, followed by a register as the operand, corresponding to one of the following seven opcodes (59 | 5A | 5B | 5C | 5D | 5E | 5F). The next part that can be detected is the decoding part. This is always a XOR instruction which has an opcode of 31. In this example, this is a XOR operation with the EAX register, which can have one of the seven opcodes from 40 to 47. This is the register where the key is stored. The final byte in the rule is the location of the instruction of the shellcode that will be decoded with the XOR instruction, which is the value of the register where the instruction pointer is stored plus an offset that results in the memory location of the next
set of 4 bytes of the encoded payload. The square brackets in between the opcodes can be any arbitrary sequence of zero to the indicated number of bytes. These bytes indicate instructions like emptying and storing the loop counter in a counter register, which could be achieved with multiple instructions or sequences of instructions.

| B8 ?? ?? ?? ?? | 0-30 | D9 74 24 F4 | 0-10 | 59 | 5A |
| 5B | 5C | 5D | 5E | 5F |
| 0-50 | 41 | 40 | 41 |
| 42 | 43 | 45 | 46 | 47 |

Listing 1: Example of a single YARA rule

Looking at the names of the rules in Listing 2 in Appendix A we see that there are two conditions. In the first condition, the rule starts with the instruction to store the decoding key in a register, followed by the floating-point unit instruction to retrieve the value of the instruction pointer. In the second condition, the rule starts with the floating-point unit instruction to retrieve the value of the instruction pointer, followed by the instruction to store the decoding key. Both conditions consist of six rules that correspond to a decoder stub that XOR with a specific register.

IV. METHODOLOGY

This section describes our methodology for modifying the Shikata Ga Nai decoder stub to bypass or decrease static detection. The proposed method can also be applied to any shellcode other than that of the Shikata Ga Nai encoder.

A. Evasion Techniques

The current version of the Shikata Ga Nai encoder generates a decoder stub that can be detected with common static detection rules [1] [2]. The decoder stub must be altered to bypass the binary pattern matching of the rules. This must be done while retaining the functionality of the decoder stub.

We have used the following techniques to alter the x86 decoder stub and allow evasion of static detection:

- Polymorphism
- Encoding

Polymorphism is applied, i.e., using instruction substitutions, adding or removing instructions while retaining the original functionality. It may be required to alter the logic of the decoder stubs, e.g., adding custom JMP instructions to correct the offsets between the decoder stub and the encoded payload. Instruction substitution is described in Section II-C and can be used on one or more x86 instructions. We have found alternatives for instructions by reading online x86 instruction references that are derived from the Intel® 64 and IA-32 Architectures Software Developer’s Manual and the manual itself [7].

Encoding is applied, i.e., using alternative machine code encoding that has the same x86 mnemonic and result, e.g., instructions encoded in a different base of machine code.

To find the alternative instruction encoding for a specific instruction, we have used an open-source tool that we will discuss in the following subsection. It is important to note that this encoding technique differs from the encoding technique that is commonly seen in shellcode generators. This type of encoding builds on the fact that Intel documents alternative encoding schemes, e.g., different displacements and ModR/M addressing forms in its manual [7].

B. Bypassing Static Detection

We know that the Shikata Ga Nai decoder stub can be detected with the rule that was written by Steven Miller [1]. The rule works by detecting the binary pattern of four specific actions of the decoder stub:

1) The MOV instruction to the decoding key (opcode for a specific register plus 4 random bytes)
2) The FPU instruction to push the instruction pointer (always the same opcode of 4 bytes)
3) The POP instruction to store the instruction pointer in a register (seven possible opcodes of 1 byte)
4) The XOR instruction (first byte of opcode is always the same)

To bypass or decrease detection, we need to focus on the decoder stub’s functionality, i.e., the four listed actions. For example, if the key can be stored in another way instead of using the MOV instruction, this would already bypass the detection using the rule shown in Listing 2. All our experiments were performed on a Kali Linux [8] x86 virtual machine.

C. Lab Setup

In our research, we have used multiple tools to accomplish our goals. An overview of the tools is given in the following list:

- Shellcode Assistant
- GHex Hex Editor
- Metasploit’s msfvenom
- Irasm
- Evan’s Debugger

1) Obtaining the Raw Shellcode File: To begin our experiments, we wrote a simple Hello World program using x86 Intel assembly, which is shown in Appendix D in Listing 5. We retrieved the escaped hex string of the assembly instructions using our custom tool Shellcode Assistant [9], which was written in Python 3 for use during our experiments. We could use the Linux command echo –e to output the escaped hex string as raw data to a file, what we describe as a raw shellcode file. Instead, we piped the escaped hex string to the msfvenom utility, which is part of Rapid7’s Metasploit Framework [10], to encode the instructions with the Shikata Ga Nai encoder, as shown in Appendix C in Listing 4, putting the encoded instructions to a raw shellcode file. Raw shellcode files only contain the
x86 code that is required to execute specific sequences of instructions, i.e., a file without file headers.

2) Shellcode Assistant: Our tool Shellcode Assistant [9] offers various options that may assist a user in writing or modifying shellcode, such as an interactive command-line interface, which includes an assembler and disassembler in one. This is especially useful since it does not require switching between assembling or disassembling, which we often do while modifying the x86 instructions. In addition, our tool allows dumping instructions or raw shellcode files as escaped hex strings or x86 assembly instructions, while specifying custom offsets in the disassembled view. It can also assemble or disassemble multiple instructions, as shown in Appendix C in Listing 3 and 4. It allows compiling raw shellcode files, auto-recompilation upon the modification and producing graphs by emulating the compiled shellcode using libemu. It also offers the option to determine if a compiled shellcode file executes properly by checking the Linux standard output and standard error for a specific output string upon execution. Perhaps the most useful feature during our research is the feature that allows scanning the raw shellcode file using a YARA rule. This feature can also be utilised when using the automatic recompiling mode of our tool. That way, every time the raw shellcode file is modified on disk, it is disassembled, recompiled, tested and scanned for detection. The features have all been implemented with the thought of making the process of manually writing or modifying shellcode using a hex editor a faster and less cumbersome process. An example of the output of our tool can be seen in Appendix C in Listing 9.

3) Modifying the Shellcode: We have used a popular hex editor named Ghex [11] to create or modify raw shellcode files. With our proposed method, any hex editor could be used that supports deleting, inserting or replacing bytes and jumping to specific offsets in a file.

As mentioned earlier, we used x86 references to find instructions for applying polymorphism to the raw shellcode file. This was a manual process which required critical thinking on how to achieve the same result while using different instructions. As of the time of writing, we were unable to find any tools that automated x86 polymorphism on single instruction, raw shellcode files or hex strings.

To obtain alternative encoding for machine code, we used an open-source tool named irasm [12], which is written in Ruby and offers an interactive command-line interface, where x86 instructions could be entered to output alternative encoded machine code, i.e. alternative x86 instructions. The author of the tool XlogicX goes into detail on this subject during the presentation of his tool at DEFCON-25 [3]. The tool provides multiple valid representations for a large number of x86 instructions.

4) Locating and Fixing Issues: We utilised a tool named Evan’s debugger [13], also known as edb, to set breakpoints, modify instructions and step through our compiled shellcode to get a better understanding of the Shikata Ga Nai’s decoder stub execution. Adding a debugger to our arsenal made it easier to locate and fix issues, e.g., incorrect offsets of instructions in the decoder stub. We initially tried to use a debugger for the entire process. However, this was not possible due to the fact that debuggers usually do not allow inserting bytes and shifting code in a compiled program, as it is considered unwanted behaviour while debugging applications. Shellcode is an exception, as it is usually not dependent on its position in an application, rather it depends on the calculation of offsets and the space that can be used for exploitation, such as buffer sizes during buffer overflows or code caves during the backdooring binaries.

D. Detection Rules / Programs

To test the effectiveness of our modifications, the raw shellcode file of the encoded Hello World program with the modified decoder stub was tested against multiple static detection systems. The rules to detect Shikata Ga Nai encoded shellcode are presented in Appendix C in Listing 2, 3 and 4.

We have performed tests using our modified Shikata Ga Nai decoder against the following systems:

- YARA FireEye Rule (2019)
- Clam AV (2020-05-25)
- Cisco Firepower IDS (2020-04-27)
- Snort IDS (2020-05-21)

V. Results

This section consists of three subsections, one containing a description of the patterns that are used to detect the x86 instructions that are used by the Shikata Ga Nai decoder stub. The other containing a description of the modifications that can be applied to bypass static detection. In the final section, we present the detection results of each tested detection system.

Our proposed method was successful in bypassing various static detection systems and has multiple practical applications. As previously mentioned, it can be applied to any shellcode and is not limited to the decoder stubs of Shikata Ga Nai shellcode. We believe our findings will aid researchers in writing or modifying existing shellcode to bypass static detection.

The tested detection systems detect Shikata Ga Nai shellcode based on rules containing binary patterns, as previously shown in Section III. To determine the required modifications to bypass detection, we have split the previously mentioned YARA detection rule in four parts. This gave us a clear overview of what functionalities we need to retain while replacing x86 instructions. The Shikata Ga Nai encoder utilises dynamic code block ordering. This means that it can generate a decoder stub that first stores the initial decoding key in a random register and then retrieves the location of the instruction pointer second or that it can do this the other way around.
For the reader to be able to interpret the results correctly, we present the following table as a quick reference for the notation that is used extensively in the following subsections.

<table>
<thead>
<tr>
<th>Value</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildcard</td>
<td>??</td>
<td>Any single byte, e.g., 90</td>
</tr>
<tr>
<td>Wildcard Range</td>
<td>[0-10]</td>
<td>Range of 0 to 10 wildcard bytes</td>
</tr>
<tr>
<td>OR</td>
<td>(59</td>
<td>5A)</td>
</tr>
<tr>
<td>x86 Register</td>
<td>r32</td>
<td>A 32-bit register, e.g., EAX</td>
</tr>
<tr>
<td>x86 Pointer</td>
<td>m32</td>
<td>A 32-bit memory location, e.g., [EAX]</td>
</tr>
</tbody>
</table>

**TABLE I: Notation reference**

A. Detected x86 Instructions

Taking Shikata Ga Nai’s dynamic code block ordering and use of random registers into consideration, detection is possible using twelve distinctive patterns. The x86 instructions that were caught by these patterns are described in the Tables [II][III][IV][V][VI] and [VII]

<table>
<thead>
<tr>
<th>Step</th>
<th>Mnemonic</th>
<th>Detected Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>MOV EAX,&lt;key&gt;</td>
<td>B8 ?? ?? ?? ?? [0-30]</td>
</tr>
<tr>
<td>2/1</td>
<td>FSTENV</td>
<td>D9 74 24 F4 [0-10]</td>
</tr>
<tr>
<td>3</td>
<td>POP &lt;r32&gt;</td>
<td>(59</td>
</tr>
<tr>
<td>4</td>
<td>XOR &lt;m32&gt;,EAX</td>
<td>31 (40</td>
</tr>
</tbody>
</table>

**TABLE II: Key stored in EAX register**

<table>
<thead>
<tr>
<th>Step</th>
<th>Mnemonic</th>
<th>Detected Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>MOV EBX,&lt;key&gt;</td>
<td>B8 ?? ?? ?? ?? [0-30]</td>
</tr>
<tr>
<td>2/1</td>
<td>FSTENV</td>
<td>D9 74 24 F4 [0-10]</td>
</tr>
<tr>
<td>3</td>
<td>POP &lt;r32&gt;</td>
<td>(59</td>
</tr>
<tr>
<td>4</td>
<td>XOR &lt;m32&gt;,EBX</td>
<td>31 (58</td>
</tr>
</tbody>
</table>

**TABLE III: Key stored in EBP register**

<table>
<thead>
<tr>
<th>Step</th>
<th>Mnemonic</th>
<th>Detected Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>MOV ECX,&lt;key&gt;</td>
<td>B9 ?? ?? ?? ?? [0-30]</td>
</tr>
<tr>
<td>2/1</td>
<td>FSTENV</td>
<td>D9 74 24 F4 [0-10]</td>
</tr>
<tr>
<td>3</td>
<td>POP &lt;r32&gt;</td>
<td>(58</td>
</tr>
<tr>
<td>4</td>
<td>XOR &lt;m32&gt;,ECX</td>
<td>31 (48</td>
</tr>
</tbody>
</table>

**TABLE IV: Key stored in ECX register**

<table>
<thead>
<tr>
<th>Step</th>
<th>Mnemonic</th>
<th>Detected Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>MOV EDL,&lt;key&gt;</td>
<td>BF ?? ?? ?? ?? [0-30]</td>
</tr>
<tr>
<td>2/1</td>
<td>FSTENV</td>
<td>D9 74 24 F4 [0-10]</td>
</tr>
<tr>
<td>3</td>
<td>POP &lt;r32&gt;</td>
<td>(58</td>
</tr>
<tr>
<td>4</td>
<td>XOR &lt;m32&gt;,EDI</td>
<td>31 (78</td>
</tr>
</tbody>
</table>

**TABLE V: Key stored in EDI register**

<table>
<thead>
<tr>
<th>Step</th>
<th>Mnemonic</th>
<th>Detected Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>MOV EDX,&lt;key&gt;</td>
<td>BA ?? ?? ?? ?? [0-30]</td>
</tr>
<tr>
<td>2/1</td>
<td>FSTENV</td>
<td>D9 74 24 F4 [0-10]</td>
</tr>
<tr>
<td>3</td>
<td>POP &lt;r32&gt;</td>
<td>(58</td>
</tr>
<tr>
<td>4</td>
<td>XOR &lt;m32&gt;,EDX</td>
<td>31 (50</td>
</tr>
</tbody>
</table>

**TABLE VI: Key stored in EDX register**

B. Modifications

As previously mentioned in Section [V], we have primarily used two techniques for evading detection, i.e., polymorphism and encoding. For each part of the rule, we present one or more polymorphic or encoded alternatives. Examples of polymorphic modifications for each part of the decoder stub are shown in Appendix [C] in Listing [11][12][13][14] and [14]

The first binary pattern that needed to be modified, was that of the MOV instruction, the first step where the decoding key gets stored into a random register. As previously mentioned, this key is later used in the XOR loop to decode the actual payload. We have found and tested two alternatives for moving a 4-byte immediate value into a register, which can be seen in Table [VIII]. The polymorphic alternative first pushes the key on the memory stack and then pops it off the stack back into a register, thus avoiding the use of the MOV instruction. This modification adds 1 byte to the size of the code. The encoded alternative utilises the ModR/M byte to specify the instruction operands and their addressing mode. This allows us to use a valid alternative of the MOV instruction, as it does not have the same binary pattern as the original instruction. This modification also increases the shellcode’s length by 1 byte.

The second binary pattern that was modified, was that of the FNSTENV instruction, which stores the floating-point unit key on the memory stack and then pops it off the stack back into a register. This key is later used in the XOR loop to decode the actual payload. We have found and tested two polymorphic and one encoded alternatives, which can be seen in Table [IX]. The x86 CALL instruction stores the current location of the instruction pointer on the top of the stack and then jumps to the location given in the operand. Jumping 5 bytes ahead will save the instruction pointer on the stack and jump to the next instruction, continue executing subsequent instructions, since the CALL instruction is 5 bytes in length. However, we often would like to avoid null-bytes in specific cases, e.g., buffer overflow attacks where the null-byte might cause issues. As a workaround, we use the CALL instruction with an offset of 4 bytes instead of 5 bytes. This gives us a CALL instruction without null-bytes, but lets the program jump to the end of its own current CALL instruction, which is FF. This instruction is garbage on its own and breaks the execution of the code. We could fix this by using a ModR/M encoded instruction that would use the FF byte, such as INC EBP, written as 0xFEC5 instead of 0x45. This messes up the output of disassemblers, such as capstone [14] and ndisasm [15]. Emulation with
libemu [16] was also broken at this stage. Nevertheless, the functionalities were retained and the shellcode was executed as required. The encoded increment isn’t used by the decoder stub and is solely used to ensure that the program continues its execution. These alternatives increase the length of the decoder stub by 1 and 3 bytes respectively. Encoding the CALL and INC <r32> instruction, can also be encoded using WORD sized and Modr/M alternative encoding. This option increases the length of the decoder stub by 2 bytes.

The third binary pattern that needed to be avoided was that of the POP instructions, which in this case popped the location of the instruction pointer off the stack into a random register. Again, we have found 2 polymorphic and 2 encoded alternatives for retrieving the instruction pointer from the stack, which can be seen in Table IX We could use the MOV instruction to simply copy the 4-byte instruction pointer from the stack into a register. Another option is using the XCHG instruction, which exchanges the content of two given operands. Both of these alternatives increase the decoder stub’s size by two bytes. The encoding options make use of Modr/M and different displacement for the POP and MOV instructions respectively. This increases the length of the decoder stub by 1 and 6 bytes.

The fourth and last detected pattern consisted of a XOR instruction followed by a byte representing a combination of the memory location of the encoded payload and a random register, which holds the decoding key loaded by the initial MOV instruction, followed by another byte determining the offset of the encoded payload, which is always either 15, 19, 20 or 24, depending on the conditions that are chosen by the Shikata Ga Nai encoder script. This was the hardest part, since there is no simple alternative instruction for performing a bitwise XOR operation, with a memory location and register, while using the memory location as a destination.

We solved this issue with a workaround, first exchanging the contents of the register and memory address and then performing the bitwise XOR operation that uses the register as a destination address instead of the memory address and then exchanging the contents of the register again, which can be seen in Table X This uses the hex code 33 instead of 31, which is detected by the Shikata Ga Nai detection rules. To make things even more difficult, Shikata Ga Nai’s decoder stub needs to decode its own last instruction, i.e., the LOOP -9 instruction, on the first iteration of the XOR loop. The LOOP instruction performs a jump to the location given in the operand and decrements the CL register until it is zero. In this case, the location is the given offset, which is relative to the following instruction. The encoded value of the loop instruction is non-deterministic, since we do not know ahead of time what decoding key will be chosen by the encoder script. We chose to leave this XOR encoded LOOP instruction as it is, for it to remain script-able with the given alternatives. The LOOP instruction of Shikata Ga Nai decoder stubs always jump back 9 bytes and because we have altered the decoding loop by inserting multiple instructions, it jumps back to an incorrect offset backwards. We fixed this issue by implementing JMP and NOP instructions. The LOOP instruction now jumps to a JMP instruction, which jumps back
once again to the beginning of the decoder loop, which starts with incrementing the random register that held the instruction pointers value by 4, so that it can continue decoding the next 4 bytes of the encoded payload. On subsequent iterations, the JMP instruction that the LOOP instruction lands on is skipped using another JMP instruction to prevent never-ending loops. The LOOP instruction’s last jump backwards occurs after the CL register decrements to zero. This polymorphic alternative increased the decoders stub length by 12 bytes.

When static detection methods are known, like the YARA rule in Appendix [A] in Listing [2], it is possible to substitute binary patterns for other binary patterns that are not included in the static detection method. The choice between using polymorphism or alternative encoding is dependent on the availability of alternative instructions and the length of these. The length of encoded instructions are usually shorter in length.

Updating static detection rules by taking polymorphism and alternative encoded instructions into account may increase the detection rate of static detection systems. However, since polymorphism and alternative encoding can be used to slightly modify a decoder stub, the detection and modification of decoders will remain to be a cat and mouse game.

VI. DISCUSSION

As shown in the results in Section [V], we managed to bypass all static detection systems that we experimented with using our modified shellcode. This shows that it is possible to bypass static detection by applying polymorphism or using alternative encoded machine code.

VII. CONCLUSION

We first looked at what techniques are used in the Shikata Ga Nai decoder to bypass static detection. These techniques were instruction substitution, reordering of instructions and the use of random registers. Furthermore, we looked at how a decoder could be detected using static analysis, i.e., the Shikata Ga Nai decoder stub. Given the YARA rule from FireEye [1], four specific parts of the decoder stub were detected. These are the MOV instruction for storing the decoding key, the FNSTENV instruction to push the floating-point unit environment on the stack, the POP instruction to pop the value of the instruction pointer of the stack and the XOR instruction, which is used to decode the encoded payload with the decoding key from the first part.

Knowingly that the decoder stub can be detected by those four instructions, we substituted these instructions by applying the use of polymorphism or the use of alternative encoded instructions. For instructions like the MOV and POP instructions, both polymorphic and encoded alternatives were available. For an instruction like the XOR instruction, no direct alternative instructions were available. Even in such cases it is still possible to retain the functionality of shellcode after making modifications, as we have demonstrated.

Furthermore, our custom tool named Shellcode Assistant was created to make the process of creating or modifying shellcode, such as that of the decoder stub, a less tedious and a more intuitive process. Combined with a hex editor, this resulted in a method for easy modification or writing of shellcode. The automatic recompilation mode and the interactive command-line interface, which supports two-way assembling and disassembling, stand out in making the process of modifying shellcode to bypass static detection a faster and easier process.

In our research, we tried to find a method that allows decreasing the detection of shellcode decoders during static analysis. Packing our arsenal with our Shellcode Assistant tool and a hex editor resulted in a proposed method that allowed bypassing all of the tested static detection systems.
Most importantly, our method can also be applied to shellcode other than that of what was generated by the Shikata Ga Nai encoder.

VIII. FUTURE WORK

We are confident that our research will serve as a base for future studies on evading static detection systems. First of all, other polymorphic alterations could be used when modifying the decoder stub. For example, the use of the MMX instruction set is something that can be done to extend our research. Furthermore, our suggestions could be implemented in Shikata Ga Nai’s encoder, resulting in more permutations for the decoder stub. Another idea for future work would be to automate the modifications of the decoder stub after it has been generated. In this way, the automated modification could also be applied to other decoder stubs than Shikata Ga Nai.

REFERENCES

Fig. 1: Control-flow graph of a payload encoded with the Shikata Ga Nai encoder.
Fig. 2: Another control-flow graph Image of a payload encoded with the Shikata Ga Nai encoder.
rule Hunting_Rule_ShikataGaNai
{
  meta:
    author = "Steven Miller"
  strings:
    $varInitializeAndXorCondition1_XorEAX = { B8 ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition1_XorEBP = { BD ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition1_XorEBX = { BB ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition1_XorECX = { B9 ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition1_XorEDI = { BF ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition1_XorEDX = { BA ?? ?? ?? ?? [0-30] D9 74 24 F4 [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorEAX = { D9 74 24 F4 [0-30] B8 ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorEBP = { D9 74 24 F4 [0-30] BD ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorEBX = { D9 74 24 F4 [0-30] BB ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorECX = { D9 74 24 F4 [0-30] B9 ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorEDI = { D9 74 24 F4 [0-30] BF ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
    $varInitializeAndXorCondition2_XorEDX = { D9 74 24 F4 [0-30] BA ?? ?? ?? ?? [0-10] ( 59 | 5A | 5B | 5C | 5D | 5E | 5F ) [0-50] 31 ( 40 | 41 | 42 | 43 | 45 | 46 | 47 ) ?? }
  condition:
    any of them
}

Listing 2: YARA rule to detect the Shikata Ga Nai decoder stub

VIRUS NAME: Win.Trojan.MSShellcode-6360729-0
TDB: Engine:81-255,Target:0
+- OFFSET: ANY
+- SIGMOD: NONE
+- DECODED SUBSIGNATURE:
  +- REEGEX: ^\xd9\x74\x24\xf4[^\x50-\x5f]$\xb0-\xbf].(0,20)\{|\x29\x2b\x31\x33].(0,500)\x99\xb1.\{0,8}\x31.\{\x10-\x1f\x0f].\{|\x03\x13\x23\x33\x43\x53\x63\x73\x83\x93\xa3\xb3\xc3\xd3\xe3\xf3
  +- CFLAGS: s
}

Listing 3: Clam AV rule to detect the Shikata Ga Nai decoder stub

alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"INDICATOR-SHELLCODE_Shikata_Ga_Nai_x86_polymorphic_shellcode_decoder_detected"; c; classtype:shellcode-detect; metadata:engine shared, soid 3|17775, policy max-detect-ips drop; )
}

Listing 4: Cisco Firepower / Snort rule to detect the Shikata Ga Nai decoder stub
APPENDIX C

global _start

section .text

_start:

; print hello world on the screen
    xor eax, eax          ; Sets EAX register to 0
    mov al, 0x4           ; Moves print syscall number into AL register
    xor ebx, ebx          ; Sets EBX register to 0
    mov bl, 0x1           ; Sets file descriptor to 1 (STDOUT)
    xor edx, edx          ; Sets EDX register to 0
    push edx              ; Pushes the value of EDX onto the stack
    push 0x0a646c72        ; Pushes 'dlr' onto the stack
    push 0x6f57206f        ; Pushes 'oW' onto the stack
    push 0x6c6c6548        ; Pushes 'lleH' onto the stack
    mov ecx, esp           ; Moves the stack pointer value into the ECX register
    mov dl, 12             ; Moves amount of bytes into the DL register
    int 0x80               ; Syscall interrupt

; exit the program gracefully
    xor eax, eax          ; Empty EAX register
    mov al, 0x1           ; Moves exit syscall number into AL register
    xor ebx, ebx          ; Empty EBX register (return value)
    int 0x80               ; Syscall interrupt

Listing 5: "Hello World" x86 ASM code

Listing 6: "Hello World" x86 ASM code on one line for Shellcode Assistant

Listing 7: "Hello World" x86 escaped hex string

Listing 8: "Hello World" escaped hex string encoding by piping echo -e output with msfvenom
ASM Code:

Offset  Size  Hex       Opcode Operand
0x0      5  BB AC CB 77 92  mov   ebx, 0x9277cbac
0x5      2  DA CA      fcmov  st(0), st(2)
0x7      4  D9 74 24 F4  fnstenv    dword ptr [esp - 0x4]
0xb      1  5D         pop    ebp
0xc      2  31 C9      xor    ecx, ecx
0xe      2  B1 0B      mov    cl, 0xb
0x10     3  B3 C5 04   add    ebp, 4
0x13     3  31 5D 0F   xor    dword ptr [ebp + 0x1f], ebx
0x16     3  03 5D A3   add    ebx, dword ptr [ebp + 0x5d]
0x19     6  29 82 A3 7B 1D 69  sub    dword ptr [edx + 0x691d7ba3], eax
0x1f     1  F5         cmc
0x20     5  83 C5 04 73 C4 84  mov   al, byte ptr [0x84c473ed]
0x25     1  40         inc    eax
0x26     2  1C 54      sbb    al, 0x54
0x28     1  44         inc    esp
0x29     2  01 D6      add    esi, edx

Program Execution:
STDOUT: Hello World
STDERR:

Test String = Hello World
TEST SUCCESSFUL

YARA Scan:
Detection: True
Rule: Hunting_Rule_ShikataGaNai
String: $varInitializeAndXorCondition1_XorEBX
Offset: 0

Last Modified: Sun May 31 06:43:05 2020

Shellcode as hex:
0x BB AC CB 77 92 DA CA D9 74 24 F4 5D 31 C9 B1 0B 83 C5 04 31 5D 0F 03 5D A3 29 82 A3 7B 1D 69 F5 A0 ED 73 C4 84 40 1C 54 44 01 D6 F0 FB E9 B1 6F 6B A2 58 1C 07 BB 42 6E DB 76 04 BE 23 38 04 F0 78 F4 87 F8

Shellcode as escaped hex string:
\xBB\xAC\xCB\x77\x92\xDA\xCA\xD9\x74\x24\xF4\x5D\x31\xC9\xB1\x0B\x83\xC5\x04\x31\x5D\x0F\x03\x5D\xA3\x29\x82\xA3\x7B\x1D\x69\xF5\xA0\xED\x73\xC4\x84\x40\x1C\x54\x44\x01\xD6\xF0\Fx\x9F\xB1\x6F\xB2\x58\x1C\x07\xBB\x42\x6E\xDB\x76\x04\xBB\xB3\x38\x04\xF0\x78\xF4\x87\xF8

Shellcode emulated:
Graph saved as 'testing-1.bin.dot' & 'testing-1.bin.png'

Listing 9: Shellcode Assistant showcase
## Appendix D

**Filename:** testing.bin  
**Length:** 68 bytes

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Hex</th>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>5</td>
<td>BB AC CB 77 92</td>
<td>mov</td>
<td>ebx, 0x9277cbac</td>
</tr>
<tr>
<td>0x5</td>
<td>2</td>
<td>DA CA</td>
<td>fcmovet</td>
<td>st(0), st(2)</td>
</tr>
<tr>
<td>0x7</td>
<td>4</td>
<td>D9 74 24 F4</td>
<td>fnstenv</td>
<td>dword ptr [esp - 0xc]</td>
</tr>
<tr>
<td>0xb</td>
<td>1</td>
<td>5D</td>
<td>pop</td>
<td>ebp</td>
</tr>
<tr>
<td>0xc</td>
<td>2</td>
<td>31 C9</td>
<td>xor</td>
<td>ecx, ecx</td>
</tr>
<tr>
<td>0xe</td>
<td>2</td>
<td>B1 0B</td>
<td>mov</td>
<td>cl, 0xb</td>
</tr>
<tr>
<td>0x10</td>
<td>3</td>
<td>83 C5 04</td>
<td>add</td>
<td>ebp, 4</td>
</tr>
<tr>
<td>0x13</td>
<td>3</td>
<td>31 5D 0F</td>
<td>xor</td>
<td>dword ptr [ebp + 0xf], ebx</td>
</tr>
<tr>
<td>0x16</td>
<td>3</td>
<td>03 5D A3</td>
<td>add</td>
<td>ebx, dword ptr [ebp - 0x5d]</td>
</tr>
<tr>
<td>0x19</td>
<td>6</td>
<td>29 82 A3 7B 1D 69</td>
<td>sub</td>
<td>dword ptr [edx + 0x691d7ba3], eax</td>
</tr>
<tr>
<td>0x1f</td>
<td>1</td>
<td>5F</td>
<td>cmc</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>1</td>
<td>A0 ED 73 C4 84</td>
<td>mov</td>
<td>al, byte ptr [0x84c473ed]</td>
</tr>
<tr>
<td>0x25</td>
<td>1</td>
<td>40</td>
<td>inc</td>
<td>eax</td>
</tr>
<tr>
<td>0x26</td>
<td>2</td>
<td>1C 54</td>
<td>sbb</td>
<td>al, 0x54</td>
</tr>
<tr>
<td>0x28</td>
<td>1</td>
<td>44</td>
<td>inc</td>
<td>esp</td>
</tr>
<tr>
<td>0x29</td>
<td>2</td>
<td>01 D6</td>
<td>add</td>
<td>esi, edx</td>
</tr>
</tbody>
</table>

---

**Program Execution:**  
**STDOUT:** Hello World  
**STDERR:** Test String = Hello World  
**TEST SUCCESSFUL**

---

### Listing 10: Test 0 - Untouched Shikata Ga Nai encoded "Hello World"

**Filename:** testing.bin  
**Length:** 69 bytes

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Hex</th>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>5</td>
<td>68 AC CB 77 92</td>
<td>push</td>
<td>0x9277cbac</td>
</tr>
<tr>
<td>0x5</td>
<td>1</td>
<td>5B</td>
<td>pop</td>
<td>ebx</td>
</tr>
<tr>
<td>0x6</td>
<td>2</td>
<td>DA CA</td>
<td>fcmovet</td>
<td>st(0), st(2)</td>
</tr>
<tr>
<td>0x8</td>
<td>4</td>
<td>D9 74 24 F4</td>
<td>fnstenv</td>
<td>dword ptr [esp - 0xc]</td>
</tr>
<tr>
<td>0xc</td>
<td>1</td>
<td>5D</td>
<td>pop</td>
<td>ebp</td>
</tr>
<tr>
<td>0xd</td>
<td>2</td>
<td>31 C9</td>
<td>xor</td>
<td>ecx, ecx</td>
</tr>
<tr>
<td>0xf</td>
<td>2</td>
<td>B1 0B</td>
<td>mov</td>
<td>cl, 0xb</td>
</tr>
<tr>
<td>0x11</td>
<td>3</td>
<td>83 C5 04</td>
<td>add</td>
<td>ebp, 4</td>
</tr>
<tr>
<td>0x14</td>
<td>3</td>
<td>31 5D 0F</td>
<td>xor</td>
<td>dword ptr [ebp + 0xf], ebx</td>
</tr>
<tr>
<td>0x17</td>
<td>3</td>
<td>03 5D A3</td>
<td>add</td>
<td>ebx, dword ptr [ebp - 0x5d]</td>
</tr>
<tr>
<td>0x1a</td>
<td>6</td>
<td>29 82 A3 7B 1D 69</td>
<td>sub</td>
<td>dword ptr [edx + 0x691d7ba3], eax</td>
</tr>
<tr>
<td>0x20</td>
<td>1</td>
<td>5F</td>
<td>cmc</td>
<td></td>
</tr>
<tr>
<td>0x21</td>
<td>5</td>
<td>A0 ED 73 C4 84</td>
<td>mov</td>
<td>al, byte ptr [0x84c473ed]</td>
</tr>
<tr>
<td>0x26</td>
<td>1</td>
<td>40</td>
<td>inc</td>
<td>eax</td>
</tr>
<tr>
<td>0x27</td>
<td>2</td>
<td>1C 54</td>
<td>sbb</td>
<td>al, 0x54</td>
</tr>
<tr>
<td>0x29</td>
<td>1</td>
<td>44</td>
<td>inc</td>
<td>esp</td>
</tr>
<tr>
<td>0x2a</td>
<td>2</td>
<td>01 D6</td>
<td>add</td>
<td>esi, edx</td>
</tr>
</tbody>
</table>

---

**Program Execution:**  
**STDOUT:** Hello World  
**STDERR:** Test String = Hello World  
**TEST SUCCESSFUL**

---

### Listing 11: Test 1 - Modified MOV instruction which stores the decoding key

**Filename:** testing.bin  
**Length:** 69 bytes

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Hex</th>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>5</td>
<td>68 AC CB 77 92</td>
<td>push</td>
<td>0x9277cbac</td>
</tr>
<tr>
<td>0x5</td>
<td>1</td>
<td>5B</td>
<td>pop</td>
<td>ebx</td>
</tr>
<tr>
<td>0x6</td>
<td>2</td>
<td>DA CA</td>
<td>fcmovet</td>
<td>st(0), st(2)</td>
</tr>
<tr>
<td>0x8</td>
<td>4</td>
<td>D9 74 24 F4</td>
<td>fnstenv</td>
<td>dword ptr [esp - 0xc]</td>
</tr>
<tr>
<td>0xc</td>
<td>1</td>
<td>5D</td>
<td>pop</td>
<td>ebp</td>
</tr>
<tr>
<td>0xd</td>
<td>2</td>
<td>31 C9</td>
<td>xor</td>
<td>ecx, ecx</td>
</tr>
<tr>
<td>0xf</td>
<td>2</td>
<td>B1 0B</td>
<td>mov</td>
<td>cl, 0xb</td>
</tr>
<tr>
<td>0x11</td>
<td>3</td>
<td>83 C5 04</td>
<td>add</td>
<td>ebp, 4</td>
</tr>
<tr>
<td>0x14</td>
<td>3</td>
<td>31 5D 0F</td>
<td>xor</td>
<td>dword ptr [ebp + 0xf], ebx</td>
</tr>
<tr>
<td>0x17</td>
<td>3</td>
<td>03 5D A3</td>
<td>add</td>
<td>ebx, dword ptr [ebp - 0x5d]</td>
</tr>
<tr>
<td>0x1a</td>
<td>6</td>
<td>29 82 A3 7B 1D 69</td>
<td>sub</td>
<td>dword ptr [edx + 0x691d7ba3], eax</td>
</tr>
<tr>
<td>0x20</td>
<td>1</td>
<td>5F</td>
<td>cmc</td>
<td></td>
</tr>
<tr>
<td>0x21</td>
<td>5</td>
<td>A0 ED 73 C4 84</td>
<td>mov</td>
<td>al, byte ptr [0x84c473ed]</td>
</tr>
<tr>
<td>0x26</td>
<td>1</td>
<td>40</td>
<td>inc</td>
<td>eax</td>
</tr>
<tr>
<td>0x27</td>
<td>2</td>
<td>1C 54</td>
<td>sbb</td>
<td>al, 0x54</td>
</tr>
<tr>
<td>0x29</td>
<td>1</td>
<td>44</td>
<td>inc</td>
<td>esp</td>
</tr>
<tr>
<td>0x2a</td>
<td>2</td>
<td>01 D6</td>
<td>add</td>
<td>esi, edx</td>
</tr>
</tbody>
</table>

---

**Program Execution:**  
**STDOUT:** Hello World  
**STDERR:** Test String = Hello World  
**TEST SUCCESSFUL**
### Listing 12: Test 2 - Modified FNSTENV instruction which stores the location of the instruction pointer

```
Filename = testing.bin
Length = 76 bytes

ASM Code:
Offset  Size  Hex     Opcode  Operand
0x00    5     68 AC CB 77 92  push 0x9277cbac
0x05    1     5B      pop   ebx
0x0A    5     E8 FF FF FF FF  call 0xe
0x0F    3     C5 5D 83  lds   ebx, ptr [ebp - 0x7d]
0x12    1     ED      in    eax, dx
0x13    2     02 31  add   dh, byte ptr [ecx]
0x15    1     C9      leave
0x16    2     B1 0B  mov   cl, 0xb
0x18    3     83 C5 04  add   ebp, 4
0x1B    3     31 5D 0F  xor   dword ptr [ebp + 0xf], ebx
0x1E    3     03 5D A3  add   ebx, dword ptr [ebp - 0x5d]
0x21    6     29 82 A3 7B 69  sub   dword ptr [edx + 0x691d7ba3], eax
0x27    1     F5      cmc
0x28    5     A0 ED 73 C4 84  mov   al, byte ptr [0x84c473ed]
0x2D    1     40      inc   eax
0x2E    2     1C 54  sbb   al, 0x54
0x30    1     44      inc   esp
0x31    2     01 D6  add   esi, edx

Program Execution:
STDOUT: Hello World
STDERR:
Test String = Hello World
TEST SUCCESSFUL
```

### Listing 13: Test 3 - Modified POP instruction which places the location of the instruction pointer into a register

```
Filename = testing.bin
Length = 74 bytes

ASM Code:
Offset  Size  Hex     Opcode  Operand
0x00    5     68 AC CB 77 92  push 0x9277cbac
0x05    1     5B      pop   ebx
0x06    5     E8 FF FF FF FF  call 0xe
0x0B    6     C5 8B 2C 24 90 90  lds   ecx, ptr [ebx - 0x6f6fdbd4]
0x11    1     90      nop
0x12    2     31 C9  xor   ecx, ecx
0x14    2     B1 0B  mov   cl, 0xb
0x16    3     83 C5 04  add   ebp, 4
0x19    3     31 5D 0F  xor   dword ptr [ebp + 0xf], ebx
0x1C    3     03 5D A3  add   ebx, dword ptr [ebp - 0x5d]
0x1F    6     29 82 A3 7B 69  sub   dword ptr [edx + 0x691d7ba3], eax
0x25    1     F5      cmc
0x26    5     A0 ED 73 C4 84  mov   al, byte ptr [0x84c473ed]
0x2B    1     40      inc   eax
0x2C    2     1C 54  sbb   al, 0x54
0x2E    1     44      inc   esp
0x2F    2     01 D6  add   esi, edx

Program Execution:
STDOUT: Hello World
STDERR:
Test String = Hello World
TEST SUCCESSFUL
```

Listing 12: Test 2 - Modified FNSTENV instruction which stores the location of the instruction pointer

Listing 13: Test 3 - Modified POP instruction which places the location of the instruction pointer into a register
### Program Execution:

STDOUT: Hello World  
STDERR: Test String = Hello World  

---

**Listing 14: Test 4 - Modified XOR instruction which is used in the XOR+additive decoding loop**