Least Information Redundancy Algorithm of Printable Shellcode Encoding for x86

Yuanding Zhou

October 6, 2023

Least Information Redundancy Algorithm of Printable Shellcode Encoding for x86

Yuanding Zhou[1]

Institute of Software Chinese Academy of Sciences yuanding2021@iscas.ac.cn

Abstract. Shellcode is a critical element in computer security that exploits vulnerabilities within software systems. Shellcode is written in machine code and often designed to be compact in size, evading detection by security software. Printable shellcode, specifically, comprises only printable ASCII characters (0x21-0x7E), including letters, numbers, and punctuation marks. The key advantage of printable shellcode lies in its ability to be embedded within data streams, which may undergo parsing or manipulation by applications that would otherwise filter or modify non-printable characters. The prevalent methods for generating printable shellcode involve encoding algorithms, such as the Riley Eller algorithm (integrated into Metasploit). However, previous research on printable shellcode has primarily focused on the availability and reduction of the encoded shellcode's size, without adequately considering the constraint imposed by the information entropy of the encoding algorithm within the context of printable shellcode. In this paper, we demonstrate the existence of minimal information redundancy in printable shellcode. Subsequently, we introduce Lycan, an implementation of a novel algorithm that surpasses previous encoding algorithms in terms of the size efficiency of the encoded shellcode. Lycan achieves the least theoretical information redundancy. Through experimentation, we observe that Lycan generates the most compact shellcode among existing tools when the shellcode's size exceeds a certain threshold.

Keywords: Shellcode · Encoding · x86

1 Introduction

Shellcode refers to a compact and executable piece of machine code that exploits vulnerabilities within a computer system, enabling the execution of arbitrary commands or the unauthorized access of the system. Typically written in machine code, shellcode is specifically crafted to be injected into a vulnerable program or system. Its small size presents a challenge for security software, as it can evade detection.

An arbitrary byte can take any value ranging from 0x00 to 0xFF, encompassing both printable and non-printable characters. The requirement for printable shellcode arises from the fact that numerous computer systems and applications tend to filter or alter non-printable characters, including control characters and null bytes. Consequently, shellcode containing such characters may be subject to filtering or modification, leading to its ineffectiveness.

One of the key advantages of printable shellcode lies in its capacity to circumvent specific security measures that aim to detect non-printable characters. For instance, certain firewalls and intrusion detection systems (IDS) are configured to monitor network traffic, searching for character sequences that match commonly employed shellcode patterns. By utilizing printable shellcode, attackers can elude these security measures, executing their malicious code without detection.

A printable character is defined as a byte ranging from 0x21 to 0x7E, encompassing a total of 94 characters in a byte's character set. While the turing-completeness of printable bytes relies on the system's Instruction Set Architecture (ISA), it is evident that the instruction set formed by printable bytes is turing-complete on x86 systems, as demonstrated by existing printable shellcode encoding algorithms [1,3,6]. In order to create printable shellcode, encoding algorithms incorporate various modifications from different perspectives, employing clever techniques.

Rix [1] introduced a technique for writing ia32 alphanumeric shellcodes. The main contribution of this work was a comprehensive compilation of all the viable alphanumeric instructions available in the ia32 Instruction Set Architecture (ISA). Moreover, Rix devised an innovative and influential approach called the XOR patching technique. This technique effectively resolved the challenge of incorporating necessary non-printable bytes within the shellcode. Rix's methodology played a pivotal role in enabling the development of ia32 alphanumeric shellcodes.

Riley Eller [3] introduced an algorithm known as the SUB encoder, which was designed to bypass MSB Data Filters. These filters are responsible for filtering out or modifying any values that fall outside the range of 0x21 to 0x7E hex in exploit code. The SUB encoder algorithm has gained significant popularity for encoding non-printable shellcode in various scenarios and has been integrated into the widely-used penetration testing tool, Metasploit. In essence, the SUB encoder utilizes the SUB instruction to convert any byte into printable bytes by performing up to three subtraction operations.

In contrast to the aforementioned algorithm, Zsolt Geczi and Péter Ivanyi [4] proposed a distinct method for converting arbitrary instructions into a printable instruction sequence. Their approach involved employing a technique known as source-to-source conversion to translate the original shellcode into an equivalent printable form. However, a notable drawback of their method is the inefficiency of the translation algorithm. It utilizes multiple redundant instructions to achieve the same effect as a single instruction, resulting in decreased efficiency.

Dhrumil Patel, Aditya Basu, and Anish Mathuria [7] have provided a comprehensive summary of the aforementioned methods and identified a significant drawback shared by all encoded shellcodes: their larger size in comparison to the original shellcodes. To address this issue, they have proposed an encoding algorithm that converts every two consecutive bytes into three printable bytes.

Additionally, they have developed a tool called "psc" that generates printable shellcode. The generated shellcode includes a runtime looped decoder, capable of transforming the encoded bytes back into the original shellcode.

In many scenarios, shellcode needs to be compact enough to fit into memory, especially in cases like buffer overflow attacks. However, in more sophisticated situations, attackers may not have the luxury of obtaining a reverse shell through direct interaction. In such cases, large-sized shellcode is necessary, such as when establishing a reverse shell via a TCP connection [15]. The existing approaches mentioned below primarily focus on generating printable shellcode by encoding the original shellcode or transforming instructions into equivalent printable machine code. Their main emphasis is on reducing the overall size of the printable shellcode. However, they tend to overlook the information redundancy introduced by the encoding algorithm, which can have a significant impact when the size of the printable shellcode exceeds a certain threshold.

In this paper, we propose a novel encoding algorithm for generating printable shellcode and analyze it from a unique perspective. We demonstrate that the least information redundancy can be achieved in the generation of printable shellcode and introduce a corresponding tool, named Lycan, that theoretically produces the most compact printable shellcode. The printable shellcode generated by Lycan includes a looped decoder written solely in printable bytes, as well as an encoded shellcode that is also printable. Our algorithm encodes every three consecutive bytes into four printable bytes, while the looped decoder transforms the printable encoded bytes back into their original form at runtime. Building upon this encoding algorithm, we present an efficient tool called Lycan, which converts the original shellcode into its printable equivalents. Lycan outperforms existing algorithms by generating the smallest printable shellcode size when the original shellcode exceeds a certain threshold. Furthermore, we have made Lycan available for public usage. Lycan and its tutorial can be found at https://github.com/zeredy879/Lycan.

Our main research contributions are as below:

- Analyze printable shellcode encoding problem from perspective of information theory and prove that the least redundancy of printable shellcode exists.
- Develop a corresponding tool called Lycan to implement the theoretically least redundancy encoding algorithm and demonstrate the feasibility in real world scenario.

The remaining sections of this paper are organized as follows. Section 2 provides an overview of the related work in the field of generating printable shellcode. Section 3 presents our findings on the existence of minimal information redundancy in printable shellcode. Section 4 outlines the details of our encoding and decoding algorithms.Section 5 elaborates on the techniques employed for writing printable shellcode and provides implementation details of our tool, Lycan. Section 6 presents the validation process and performance evaluation of Lycan. Finally, Section 7 concludes our work and summarizes the key contributions of this research.

2 Realated work

2.1 Riley Eller Algorithm

Riley Eller [3] introduced an algorithm that enables the encoding of any binary data sequence into ASCII characters. When interpreted by an Intel processor, these characters can decode the original sequence and execute it. The algorithm follows a specific process: it first moves the stack pointer just past the ASCII code, then decodes 32 bits of the original sequence at a time and pushes that value onto the stack. In summary, the Riley Eller algorithm converts arbitrary DWORD values into printable bytes by utilizing finite SUB instructions that leverage printable immediate data subtraction.

Metasploit [2] integrates the Riley Eller algorithm to SUB encoder. The SUB encoder is a dynamic polymorphic shellcode obfuscation technique, implemented as part of the Metasploit Framework. It applies a sequence of byte subtraction operations to individual shellcode values in order to generate new, non-sequential, and non-deterministic values. This encoded shellcode is observed to be stealthier compared to the original version, in terms of evading intrusion detection and prevention mechanisms. Despite the effectiveness of the SUB encoder and other Metasploit encoders in enhancing payload obfuscation, there remains a constant threat of detection and counter-measures by security systems [5]. Hence, further research is crucial to develop even more effective and efficient encoding methods for better payload protection in penetration testing and ethical hacking contexts.

The SUB encoder includes a fixed 29 byte long printable code snippet. Assume that the size of original shellcode is represented by n, the output printable shellcode's size can be calculated as $29 + 16\lceil n/4 \rceil$.

2.2 Zsolt Geczi and Peter Ivanyi's Method

Zsolt Geczi and Peter Ivanyi [4] propose a technique for automatically translating non-printable shellcodes into printable byte codes to bypass filters. Their approach, known as source-to-source conversion, involves establishing a compilation set that maps each instruction to a printable equivalent. Printable shellcode is then generated using these mapping rules. For example, the instruction 'MOV EAX, EBX' has a printable equivalent of 'PUSH EBX; POP EAX'.

While their method is context-free and convenient to extend, it suffers from a significant drawback: the resulting shellcode has a much larger size compared to existing algorithms. As a result, utilizing this output shellcode as an exploit may fail if the buffer overflow size is insufficient to accommodate the expanded exploit code.

Although the exact details of the source-to-source conversion method proposed by Zsolt Geczi and Peter Ivanyi are not publicly available, based on the example they provided [4], it can be inferred that their method generates significantly larger shellcode compared to existing works. In the only examples available, their method transformed a 38-byte shellcode into a printable shellcode of 9837 bytes. This substantial increase in size indicates that their method may not be efficient in terms of generating compact printable shellcode.

2.3 Printable Shellcode Compiler

Dhrumil Patel, Aditya Basu, and Anish Mathuria [7] propose a novel encoding scheme and a companion tool called psc (Printable Shellcode Compiler) designed to generate compact printable shellcode. One notable feature of psc is its utilization of a runtime looped decoder, similar to the approaches employed by Alpha3 [8] and Alpha Freedom [10].

In their encoding scheme, two consecutive bytes of the original shellcode are encoded into three printable bytes. During runtime, a decoding loop is employed to take these three successive bytes and transform them back into the original two bytes of the shellcode. This encoding and decoding process ensures that the generated printable shellcode is compact and retains the functionality of the original shellcode.

The psc tool includes a fixed 146 byte long decoder. Assume that the size of original shellcode is represented by n, then the output printable shellcode's size can be calculated as $146 + 3\lceil n/2 \rceil$.

3 Proof of least redundancy

3.1 The least redundancy of encoding shellcode

The original shellcode byte ranges from 0x00 to 0xFF. Assume that X represents original shellcode byte, thus the entropy of X is :

$$\begin{gathered} H(X) = -\sum_x p(x)\log_2 p(x) = \\ -\sum_{i=1}^{256} \frac{1}{256}\log_2 \frac{1}{256} = \log_2 256 = 8\ bits \end{gathered} \tag{1}$$

The printable byte ranges from 0x21 to 0x7E. Assume that E represents printable byte, thus the entropy of E is:

$$\begin{gathered} H(E) = -\sum_e p(e)\log_2 p(e) = \\ -\sum_{i=1}^{94} \frac{1}{94}\log_2 \frac{1}{94} = \log_2 94\ bits \end{gathered} \tag{2}$$

In the shellcode encoding circumstance, the fundamental encoding unit is a single byte. Assume that Y represents every encoded shellcode byte, thus the entropy of Y must be less than or equal to the entropy of E, otherwise the printable byte set is not sufficient to represent encoded bytes:

$$H(Y) \leqslant H(E) = \log_2 94\ bits \tag{3}$$

Since the H(Y) also represents the number of bits of single byte used to encode a printable character, so that H(Y) must be an integer:

$$6\ bits = \log_2 64\ bits < H(E) = \\ \log_2 94\ bits < \log_2 128\ bits = 7\ bits \tag{4}$$

Implies:

$$H(Y) \leqslant 6\ bits < \log_2 94\ bits \tag{5}$$

Since the fundamental unit of encoding algorithm is a single byte, thus assume that encoding algorithm transforms m bytes into n bytes. For information loss is not allowed during the encoding process, hence:

$$8 \times m = H(X) \times m \leqslant H(Y) \times n \leqslant 6 \times n\ bits \tag{6}$$

equivalent to:

$$\frac{m}{n} \leqslant 0.75 \tag{7}$$

The information redundancy D of encoding algorithm for printable shellcode must have:

$$D = \frac{n-m}{n} \geqslant 0.25 \tag{8}$$

3.2 Analysis

In the analysis of existing algorithms in section 2 from the perspective of information redundancy of the encoding algorithm, we observe that the SUB encoder has an information redundancy of 0.75, indicating a relatively higher level of redundancy. On the other hand, Alpha3 achieves an information redundancy of 0.5, while psc achieves a lower redundancy of 0.33. However, none of these algorithms achieve the minimum theoretically information redundancy for printable shellcode.

To address this limitation, we propose our algorithm, which aims to achieve the theoretically minimal information redundancy for generating printable shellcode. By minimizing redundancy, we can generate more compact and efficient printable shellcode. The details of our algorithm and its implications will be discussed in subsequent sections.

4 Algorithm

4.1 Encoding Algorithm

Our encoding algorithm' scheme is: encode every **3** successive bytes into **4** successive printable bytes. The detailed encoding algorithm is demonstrated as follow. The symbol '$\ll$', '$\gg$', '&' and '$\oplus$' represent left shift, right shift, logic AND, and logic XOR.

1. Check if the size of original shellcode (S) is divisible by 3, otherwise complement original shellcode with byte 0x90 (nop instruction) until the size of shellcode can be divisible by 3.
2. Take 3 successive bytes (24bits) (name them A_1 to A_3), then divide these 24 bits into 4 blocks. Every block contains 6 bits.
3. Complete each block to a single byte with 2 zero bit at significant position of a byte then form 4 bytes (name them B_1 to B_4). This step is described as following operations.
   (a) $B_1 = A_1 \gg 2$
   (b) $B_2 = ((A_1 \ll 4)\ \&\ 0x30)\ +\ (A_2 \gg 4)$
   (c) $B_3 = ((A2 \ll 2)\ \&\ 0x3C)\ +\ (A_3 \gg 6)$
   (d) $B_4 = A_3\ \&\ 0x3F$
4. Add 0x3F to each byte above. These 4 bytes (name them C_1 to C_4) constitute a group of output.This step is described as following operation.
   * $C_i = B_i\ +\ 0x3F$
5. Go back to step 2) if shellcode still has remain bytes.
6. Append 0x26 as a end token in the end of output.

The encoding process, as illustrated in Fig. 1, follows the described algorithm. Since every byte of B_1 to B_4 falls within range of 0x00 to 0x3F, thus every byte of C_1 to C_4 ranges from 0x3F to 0x7E. This range corresponds to a subset of printable ASCII characters (0x21 - 0x7E), which ensures that the encoded shellcode remains printable.

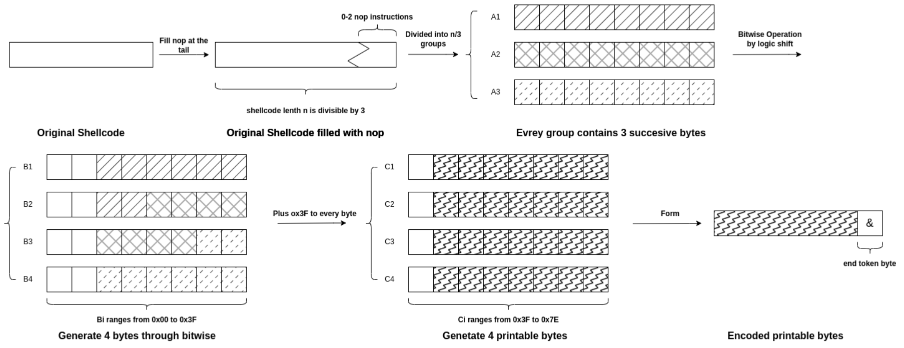
0-2 nop instructions
Fill nop at the tail
Divided into n/3 groups
A1
A2
A3
Bitwise Operation by logic shift
shellcode lenth n is divisible by 3
Original Shellcode
Original Shellcode filled with nop
Evrey group contains 3 succesive bytes
B1
B2
B3
B4
Plus ox3F to every byte
C1
C2
C3
C4
Form
&
end token byte
Bi ranges from 0x00 to 0x3F
Generate 4 bytes through bitwise
Ci ranges from 0x3F to 0x7E
Genetate 4 printable bytes
Encoded printable bytes

Fig. 1: Encoding algorithm - Every 3 successive bytes are encoded into 4 printable bytes, and each byte of encoded sequence falls within the printable ASCII range of 0x21 to 0x7E. The end token in the end of encoded sequence serves as a marker to indicate the end of the encoded sequence when decoding it. The entire sequence of encoded bytes, including the end token, remains within the printable ASCII character set, ensuring that the shellcode is fully printable.

Using the byte 0x3F as part of the encoding process offers several advantages. Firstly, 0x3F can be used as a mapping mechanism to represent the range from

0x00 to 0x3F (inclusive) within a subset of printable bytes. Secondly, by using 0x3F as a special byte within the encoding process, the decoder can more efficiently restore the encoded bytes. The convenience and effectiveness of using the 0x3F byte in the encoding algorithm will be further demonstrated and discussed in Section 4.2 and Section 5.

The end token byte 0x26 is used to mark the end of the encoded output bytes. In fact, any byte within the interval 0x21-0x3E can be used as end token, as this range is separate from the range 0x3F-0x7E, ensuring that the end token byte is distinct from the encoded bytes.

4.2 Decoding Algorithm

In correspondence with the encoding algorithm, the decoding algorithm transforms every **4** successive encoded bytes into **3** original successive bytes. The detailed decoding algorithm is demonstrated as follows.

1. Try to take 4 successive bytes (32bits) (name them C_1 to C_4) from encoded bytes. If meet the end token byte, go to step 5).
2. Add 1 to every byte of C_1 to C_4, then xor each byte with 0x3F byte. This step is described as following operation (name the output bytes B_1 to B_4).
   - $B_i = (C_i + 1) \oplus 0x3F$
3. Restore original bytes (name them A_1 to A_3) through following operations.
   (a) $A_1 = (B_1 \ll 2) \oplus (B_2 \gg 4)$
   (b) $A_2 = (B_2 \ll 4) \oplus (B_3 \gg 2)$
   (c) $A_3 = (B_3 \ll 6) \oplus B_4$
4. Bytes A_1 to A_3 constitute output, go to step 1).
5. Output bytes form the original shellcode S, and encoding process terminates.

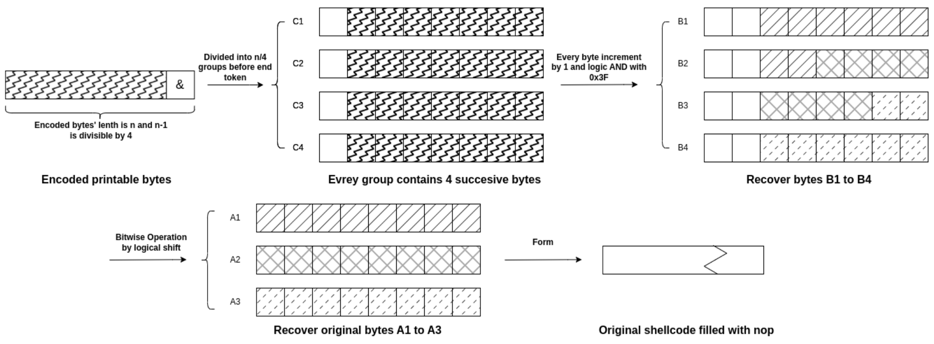
&
Encoded bytes' lenth is n and n-1 is divisible by 4
Encoded printable bytes
Divided into n/4 groups before end token
C1
C2
C3
C4
Evrey group contains 4 succesive bytes
Every byte increment by 1 and logic AND with 0x3F
B1
B2
B3
B4
Recover bytes B1 to B4
Bitwise Operation by logical shift
A1
A2
A3
Recover original bytes A1 to A3
Form
Original shellcode filled with nop

Fig. 2: Decoding algorithm

The decoding process, depicted in Figure 2, employs bitwise operations to restore the original shellcode bit by bit. To optimize the size of the decoder, we have minimized the number of bitwise operations involved. Specifically, we have combined steps 2) and 3) in the decoding algorithm implementation, resulting in a more efficient process. The implementation details of this optimization will be elaborated on in the section5.

5 Implementation

The implementation of Lycan involves two primary components: encoding and decoding. During the encoding process, the original shellcode is transformed into printable encoded bytes using the algorithm described in the encoding algorithm section. The decoding process occurs at runtime and is responsible for transforming the encoded bytes back into the original shellcode. This is achieved through a run-time printable decoder, which is implemented using handwritten x86 assembly code. The run-time decoder is designed to be printable and context-free, ensuring compatibility with any x86 shellcode.

During the encoding process, we utilize a Python script to facilitate the transformation from raw shellcode to printable form. The resulting output of the encoding process is a combination of the printable decoder and the encoded shellcode.

During the decoding process, the printable decoder is responsible for recovering the original shellcode from the encoded bytes. Once all the encoded bytes have been recovered and transformed, the program proceeds to write the decoded shellcode back into memory. This process allows us to effectively write non-printable shellcode using printable bytes and achieve the desired functionality while still maintaining the printable nature of the shellcode.

In this section we will demonstrate the process of contructing this printable decoder and explain some tricks and trade-offs used to manually write the printable decoder.

5.1 Overview of the printable decoder

Listing 1.1 describes the printable decoder architecture layout before adding xor patching instructions.

```
setup:
    pusha
    mov encoded, %esi
    mov %esi, %edi
    ...
looper:
    ...
    jne looper
_end:
    popa
encoded:
    ...
    ; encoded bytes
```

Listing 1.1: Architecture of the printable decoder. Setup section is responsible for saving the context of registers and initializing the registers used during the decoding process used for decoding, section looper contains the main loop of the decoder and is responsible for recovering original shellcode from encoded bytes, _end section is used to restore the registers' context that was saved in the setup section, encoded section is used to store docoded bytes

In order to preserve the context before the decoder runs, we use instruction PUSHA in section *setup* and POPA in section *_end* to save all the registers' context at run-time. Notice that the original shellcode is shorter than encoded bytes, so we don't need extra place to save recovered shellcode, which indicates the decoding algorithm is an In-place algorithm.

Listing 1.2 describes the setup phase of the printable decoder.

```
setup:
    pusha
    pusha
    push %eax
    pop %esi
    push $0x5E
    pop %eax
    push %eax
    pop %ecx
    xor $0x5E, %al
    push %eax
    pop %ebp
    dec %eax
    xor $0x5E, %al
    push %eax
    pop %edx
    ; edx = 0xFFFFFFA1, ecx = 0x5E, ebp = 0
```

Listing 1.2: Setup phase. This sections saves the context of registers and store special values into registers used for xor patching section 5.3.

The setup phase in the printable decoder serves the purpose of preserving the registers' context and initializing the necessary constants for the subsequent xor patching and looper phases. In this context, it is assumed that the start address of the entire shellcode is stored in register EAX [1].

In the xor patching technique, two constants, namely byte 0x5E and byte 0xA1, are required to transform arbitrary bytes into printable bytes. The selection of these specific bytes is based on their suitability for xor patching operations. Through an analysis of the entire byte set, it was determined that using the pair (0x5E, 0xA1) requires the fewest instructions for xor patching non-printable

[1] the choice of register used for marking encoded bytes can vary. Register EAX is used here as an illustration to showcase the functionality and feasibility of Lycan

bytes. Additionally, generating these constants is straightforward, as 0x5E is a printable byte and the logical XOR operation between 0x5E and 0xA1 yields 0xFF, which can be easily obtained by decrementing a zero-initialized register.

Furthermore, the constant zero value is stored in register EBP for the purpose of efficiently clearing registers to zero using the PUSH and POP instructions. This allows for a convenient and concise way to initialize registers to zero without requiring additional instructions.

Fig. 3 describes the looper phase workflow of the printable decoder.

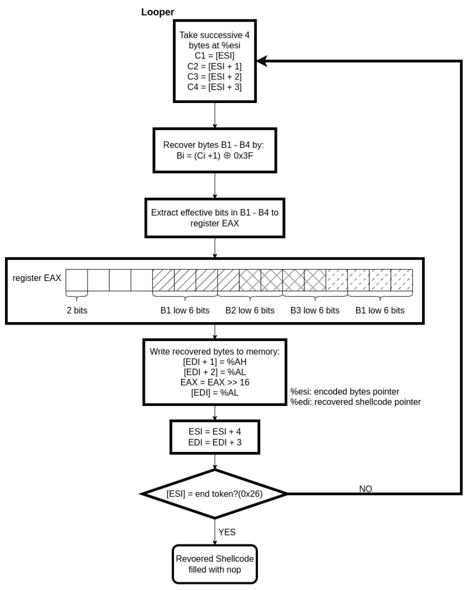

Fig. 3: Looper phase workflow. Register ESI represents read pointer of the encoded bytes, EDI represents write pointer of the recovered shellcode. After looper encounters the end token, execution flow of looper ends and jump to recovered shellcode immediately.

During the looper phase of the printable decoder, specific registers are assigned specific roles to facilitate the recovery of the original shellcode from the encoded bytes. Register ESI serves as the read pointer, indicating the location of the encoded bytes, while register EDI acts as the write pointer, indicating where the recovered shellcode will be stored. In the initial state, both ESI and EDI are set to the same value since the decoding algorithm operates in-place, and the recovery process for each group of encoded bytes is independent.

To maximize utilization of the information entropy carried by each byte, the decoding process avoids performing bitwise operations on a register multiple times. Instead, it extracts the effective bits of every four successive bytes into a single register, namely EAX. By doing so, the original shellcode can be recovered byte by byte and written back to memory. The looper iterates through each group of encoded bytes, recovering them into the original shellcode and storing them in memory until the end mark token is encountered. By recovering and storing the shellcode byte by byte, the looper phase effectively reconstructs the original shellcode from the encoded bytes using the runtime printable decoder.

The final step in making the printable decoder fully printable is the xor patching phase 5.3. This phase involves xoring every non-printable byte in the looper phase with specific values to transform them into printable bytes. To achieve this, the offset of all the non-printable bytes needs to be determined in advance, ensuring that the xor patching instructions remain printable.

Once the printable decoder has successfully recovered the complete shellcode, the control flow is transferred to the shellcode itself, allowing the original shellcode to execute. This ensures the seamless execution of the desired functionality encoded within the shellcode.

5.2 Assignment operation of register

To overcome the limitations of using non-printable instructions in the printable shellcode, we employ stack-related instructions for assignment operations. Instead of using a traditional assignment instruction like MOV, we leverage PUSH and POP instructions to achieve the desired assignment.

For instance, consider the assignment operation from register EAX to EBX. In a normal context, this would be accomplished using a MOV instruction. However, in the printable shellcode, the raw bytes of the MOV instruction contain non-printable bytes, rendering it unusable. Therefore, we resort to an alternative approach, as demonstrated in Listing 1.3, where we utilize PUSH and POP instructions to perform the assignment.

By pushing the value of EAX onto the stack and then popping it into EBX, we effectively transfer the value from one register to another. This technique allows us to accomplish assignment operations using printable instructions and ensures the compatibility of the printable shellcode with the target system.

```
push %eax ; 0x50
pop %ebx  ; 0x5B
# equivalent to ‘mov %eax, %ebx‘
```

Listing 1.3: Assuginment from EAX to EBX

When it comes to assigning arbitrary immediate data to a register, the printable shellcode faces additional challenges. While SUB encoder utilizes multiple SUB instructions to convert the zero value of a register to arbitrary 4 bytes, our decoder focuses on single-byte assignment operations rather than arbitrary 4 bytes. This is because the xor patching technique used in our decoder only operates on a single byte.

To address the assignment of specific bytes, such as 0x00 and 0xFF, to registers AL and BL, we can utilize printable instructions. Listing 1.4 demonstrates how to achieve this.

```
    push $0x50          ; 0x6A 0x50
    pop %eax            ; 0x58
    xor %al, $0x50      ; 0x34 0x50
    ; assign 0x00 to %al through
    ; xoring same data

    push %eax           ; 0x50
    pop %ebx            ; 0x5B
    dec %ebx            ; 0x4B
    ; assign 0xFF to %bl through
    ; decrement 0x00
```

Listing 1.4: Assign 0x00 and 0xFF to register AL and BL

5.3 xor patching technique

Xor patching technique, as described by Rix, is used to write non-printable instructions by XORing the non-printable byte with a printable byte in memory. To successfully perform xor patching, it is necessary to have knowledge of the address layout of the shellcode, including the start address of the shellcode and the offset of the non-printable byte that needs to be patched.

Listing 1.5 provides a template instruction for xor patching in our decoder:

```
    xor %dl, 0x3D(%esi) ; 0x30 0x56 0x3D
```

Listing 1.5: Example of xor patching technique

In the example provided in Listing 1.5, a representative xor patching instruction is shown. In the context of our decoder, the register %dl is being assigned the value 0xA1, and the value 0x3D (%esi) represents the offset between register ESI and the non-printable byte that needs to be xor patched.

The key point of xor patching is to ensure that the offset byte (0x3D) is printable. This requires careful arrangement of the offset in our decoder to ensure that it falls within the range of printable bytes (0x21-0x7E). However, this compromise on the size of the printable decoder means that we may need to include some unused bytes to ensure that the offset remains within the printable byte range. This trade-off between the size of the printable decoder and the printable range of the offset is necessary to ensure that the xor patching technique can be effectively applied to write non-printable instructions using printable bytes.

6 Evaluation

We evaluate Lycan's performance from 2 perspectives: encoding bytes length and the total bytes length. We compare Lycan with Riley Eller's SUB encoder [2], Jan Wever's Alpha3 [8] and Basu's psc [7]. Fig 4 describes the comparison of encoded bytes length and Fig 5 describes the comparison of total bytes length.

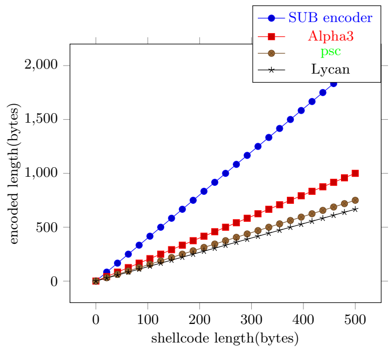

Fig. 4: Encoded bytes length

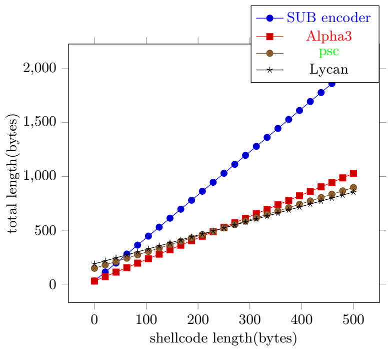

Fig. 5: Total bytes length

The encoded byte size of the different encoding algorithms exhibit a linear correlation with the size of the original shellcode, as mentioned in the previous section. Lycan, with its minimal information redundancy, outperforms the other algorithms in terms of the length of the encoded bytes, as illustrated in Figure 4 .

It is important to note that the total byte length depends on both the encoding algorithm and the length of the decoder, as all the algorithms utilize fixed-length decoders. As depicted in Figure 5 , when the length of the shellcode is less than 256, psc and Alpha3 generate shorter printable shellcode. However, when the shellcode length exceeds 256, Lycan demonstrates superior performance and produces the shortest printable shellcode.

This comparison highlights the advantage of Lycan in minimizing the length of the encoded shellcode, making it a favorable choice for scenarios where the size of the shellcode is a critical factor.

The comparison presented in Table 1 highlights the superior performance of Lycan in scenarios where the original shellcode size is significant. This makes Lycan a valuable tool for encoding and generating p rintable shellcode, especially for larger payloads.

7 Conclusion

Printable shellcode encoding algorithms are widely used to generate general shellcode in various scenarios. In this paper, we demonstrate that the least information redundancy of printable shellcode encoding algorithm is 0.75 theoretically and present corresponding algorithm which encodes 3 successive bytes to 4 bytes. Our algorithm generates the shortest encoded bytes among all the existing algorithms. Then we present Lycan – a tool that implements the algorithm. However due to unavoidable expenses of the printable decoder, the printable shellcode generated by Lycan is longer than SUB encoder and Alpha3 when the length of original shellcode is too short. Lycan's performance is best among all the algo-

Shellcode	Original	SUB encoder	Alpha3	psc	Lycan
execve(/bin/sh) [11]	20	109	68	176	216
INSERTION Encoder [12] / Decoder execve(/bin/sh)	88	381	204	278	308
OpenSSL Encrypt (aes256cbc) Files (test.txt) [13]	185	781	398	425	436
chmod 777 (/etc/passwd + /etc/shadow) + Add Root User (ALI/ALI) To /etc/passwd + Execute /bin/sh [14]	378	1549	784	713	692
Reverse (127.0.0.1 :53/UDP) Shell (/bin/sh) [15]	668	2701	1364	1148	1080

Table 1: The real world shellcode performance of distinct encoding tools

rithms when the original shellcode's size exceeds the threshold 252. In the future work, we will extend our tool to work on different ISA.

References

1. Rix. Writing IA32 alphanumeric shellcode. Phrack, 57(15), 2001. http://phrack.org/issues/57/15.html.
2. Metasploit sub encoder. https://www.rapid7.com/db/modules/encoder /x86/opt_sub/.
3. Riley Eller. Bypassing msb data filters for buffer over-flow exploits on intel platforms. http://julianor. tripod.com/bc/bypass-msb.txt.
4. Géczi, Zsolt, and Peter Iványi. "Automatic translation of assembly shellcodes to printable byte codes." Pollack Periodica 13.1 (2018): 3-20.
5. Polychronakis, Michalis, Kostas G. Anagnostakis, and Evangelos P. Markatos. "Comprehensive shellcode detection using runtime heuristics." Proceedings of the 26th Annual Computer Security Applications Conference. 2010.
6. Ding, Wenbiao, et al. "Automatic construction of printable return-oriented programming payload." 2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE). IEEE, 2014.
7. Patel, Dhrumil, Aditya Basu, and Anish Mathuria. "Automatic generation of compact printable shellcodes for x86." Proceedings of the 14th USENIX Conference on Offensive Technologies. 2020.
8. B.J. Wever. ALPHA3. https://github.com/SkyLined/alpha3

9. Mason, Joshua, et al. "English shellcode." Proceedings of the 16th ACM conference on Computer and communications security. 2009.
10. Basu, Aditya, Anish Mathuria, and Nagendra Chowdary. "Automatic generation of compact alphanumeric shellcodes for x86." Information Systems Security: 10th International Conference, ICISS 2014, Hyderabad, India, December 16-20, 2014, Proceedings 10. Springer International Publishing, 2014.
11. Linux/x86 - execve(/bin/sh) Shellcode
https://www.exploit-db.com/shellcodes/46809
12. Linux/x86 - INSERTION Encoder / Decoder execve(/bin/sh)
https://www.exploit-db.com/shellcodes/46519
13. Linux/x86 - OpenSSL Encrypt (aes256cbc) Files (test.txt) Shellcode
https://www.exploit-db.com/shellcodes/46791
14. Linux/x86 - chmod 777 (/etc/passwd + /etc/shadow) + Add Root User (ALI/ALI) To /etc/passwd + Execute /bin/sh Shellcode
https://www.exploit-db.com/shellcodes/34262
15. Linux/x86 - Reverse (127.0.0.1:53/UDP) Shell (/bin/sh) Shellcode
https://www.exploit-db.com/shellcodes/42208