Return-Oriented Vulnerabilities in ARM Executables

Zi-Shun Huang
Center for Embedded Computer Systems
University of California Irvine
zishunh@uci.edu

Ian G. Harris
Center for Embedded Computer Systems
University of California Irvine
harris@ics.uci.edu

Abstract

Return-oriented programming is a method of computer exploit technique which is growing in popularity among attackers because it enables the remote execution of arbitrary code without the need for code injection. Return-to-LibC (Ret2LibC) is the most common return-oriented attack in use today, allowing an attacker to leverage control of the stack to execute common library functions which are already present on the target system, such as LibC. ARM-based processors, commonly used in embedded systems, are not directly vulnerable to Ret2LibC attacks because function arguments in the ARM are passed through registers rather than the stack. In 2011 Itzhak Avraham presented a new Return-to-Zero-Protection (Ret2ZP) attack against ARM processors which enables the same control as a Ret2LibC attack.

Our research contribution is to provide a formal definition of the Ret2ZP attack and to define an algorithm to detect vulnerabilities to Ret2ZP in ARM executables. Our algorithm for detecting vulnerabilities can be used to screen executables for vulnerabilities before they are deployed.

1 Introduction

Within the past 15 to 20 years embedded systems have seen increasingly widespread adoption. The complexity of many embedded devices is effectively invisible to users, operating inside components used everyday including automobiles, televisions, and video game consoles. The research firm IDC reports the market for embedded computer systems already generates more than US 1 trillion in revenue annually and will double in size over the next four years [20]. IDC also predicts that much of this growth will be propelled by more sophisticated, cloud-connected embedded system which will have a high degree of network connectivity. The embedded nature of these systems successfully conceals complexity, allowing users to forget about security vulnerabilities that they are exposed to. The media has improved public awareness of cyberattacks against servers operating at institutions such as banks, credit card agencies, and nuclear material enrichment facilities. However, people are not sufficiently aware of the vulnerabilities inside embedded devices which they use much more frequently. Embedded systems are increasingly network-enabled, typically connected to the internet through a TCP/IP stack. Network connectivity makes these systems vulnerable to many of the same cyberattacks as desktop, laptop, and server machines.

Malware is a collection of instructions that executes on a machine and make the system to perform somemalicious operation [4]. There are several recent examples of malware which targets embedded devices. For example, millions of printers were found to contain a security weakness that could allow attackers to take control of the systems, steal data and issue commands that could cause the devices to overheat and catch fire [10]. Even in industrial systems such as programmable logic controller (PLC) and supervisory control and data acquisition (SCADA) have the risk to be attacked. The Stuxnet worm, discovered in June 2010, initially spreads via Microsoft Windows, and targets Siemens industrial software and equipment [13]. Embedded systems are often required to meet strict timing, cost, and power requirements. As a result, traditional defense mechanisms are not suitable for embedded systems. Security vulnerabilities often depend on very specific aspects of the behavior the underlying processor, so embedded systems vulnerabilities may be significantly different from those of general-purpose platforms.

Return-oriented programming [24] describes a growing class of exploits which code segments already present on a system to execute a range of malicious behaviors. The most common type of return-oriented

attack is Return-to-LibC (Ret2LibC) [8] where an attacker exploits a buffer overflow to redirect control flow to a library function already present in the system. To perform a Ret2LibC attack, the attacker must overwrite a return address on the stack with the address of a library function. Additionally, the attacker must place the arguments of the library function on the stack in order to control the execution of the library function. Ret2LibC is commonly applied against Intel x86 architectures, but the ARM architectures [17] used in embedded systems are not directly vulnerable to this attack because function parameters are passed through registers instead. ARM-based processors are the most commonly used microprocessors in embedded systems due in part to their low power consumption compared to Intel x86.

At the BlackHat Convention in 2011, Itzhak Avraham [6] first presented the Return-to-Zero-Protection (Ret2ZP) attack which effectively allows a Ret2LibC attack to be applied to ARM-based processors. This is an important development because reveals a significant vulnerability in ARM-based processors which is likely to be exploited to create malware targeted at embedded systems.

The contribution of this paper is an approach and a tool to analyze the software on an ARM-based system and determine whether or not it is vulnerable to a Ret2ZP attack. The feasibility of a Ret2ZP attack depends on the existence of special code sequences on the embedded system platform. We define the nature of the code sequences which expose a system to a Ret2ZP attack. Our tool automatically scans an existing ARM executable and identifies all vulnerable code sequences. Once identified, vulnerable code sequences can be patched to remove the vulnerability.

2 Related Work

Return-to-LibC (Ret2LibC) [11] is a method of exploiting based on buffer overflow to defeat a system that has a non-executable stack. The difference between Ret2LibC and traditional buffer overflow code inject attack is that the return address is overwritten to point to a shared library such as C library. In addition, the required arguments to the shared library function are also placed on the stack. This allows attackers to call existing library functions without injecting malicious code into a program. Dynamic shared libraries must be executable, so non-executable memory regions provide no protection against this attack. Ret2LibC is a special case of a return-oriented programming attack [24] which has the potential to enable arbitrary code execution.

To prevent Ret2LibC attacks, many mechanisms have conceived. One protection technique focuses on monitoring control flow and memory activities to detect violations [23]. These approaches may use hardware support [14, 5, 19, 25] such as a shadow stack. These methods require a new architecture [15, 7], and may affect the original pipeline performance [16, 3]. This techniques need to check all instructions, memory, or stack. If it does not check all instruction, the risk of attack is possible. Checking all instruction comes with a significant reduction in performance [21, 12, 9]. Protected free-branch instructions technique can be implemented by using code rewriting techniques to remove all unaligned instructions that can be used to link the gadgets. However, because of code rewriting, its main disadvantage is code size increase [22].

Ret2LibC attack requires passing arguments to libc functions from the stack. By ARM calling convention [17], passing parameters occurs through registers rather than the stack. A stack buffer overflow gives the attacker direct control of the stack, but not the registers. As a result, Ret2LibC attacks does not work againstARM-based processors. In [6], Avraham developed the Return-to-Zero-Protection (Ret2ZP) attack against ARM-based processors which allows an attacker to control registers from the stack. By allowing control of registers, Ret2ZP enables the Ret2LibC attack to be applied to ARM-based processors.

3 Return to Zero Protection (Ret2ZP)

A Ret2ZP attack exploits a stack buffer overflow to redirect control flow to a function present on the system, and to control the arguments to that function. Redirecting control flow can be performed directly by overwriting a return address on the stack with the address of the desired function. However, controlling the function arguments is more difficult because functions executing on the ARM architecture accept arguments through argument registers r0 - r3, rather than the stack. To control the arguments to a function, the contents of one or more of these registers must be assigned before control flow is redirected to the desired function.

To control the values of the argument registers and to redirect control flow to the desired function, the Ret2ZP attack depends on the existence of a **vulnerable code sequence (VCS)**, already present on the system, which copies data from the stack to the

argument registers, and then from the stack to the program counter (PC) register. The Ret2ZP attack first places the library function arguments and the library function address onto the stack. Then control flow is redirected to the VCS which moves the arguments from the stack to the argument registers, and moves the library function address from the stack to the PC, redirecting control flow to the library function.

Figures 1a, b, and c show the contents of the stack at different points during the execution of a Ret2ZP attack. Figure 1a shows the contents of the original stack, before the attack has been executed. Two stack frames are shown, where stack frame 0 is the current stack frame at the top of the stack. The stack frames have been simplified for the purposes of this presentation, so only the contents important to the Ret2ZP attack are shown. Each stack frame contains a space labeled *locals* for local variables for the corresponding function, and a space labeled *ret_addr* which contains the return address for code execution after the corresponding function is complete. The current stack frame contains a buffer labeled *buff* which we assume is vulnerable to buffer overflow by the attack.

Figure 1b shows the contents of the stack after the buffer overflow. The stack grows down in memory addresses, so the memory addresses increase down in the stack pictures of Figure 1. When the buffer overflows, the stack below the buffer is altered, as can be seen in Figure 1b. The buffer overflow places three different blocks of data onto the stack. The address of the VCS, *&VCS*, is written over the old return address so that the the VCS will be executed when the current function completes. The arguments to be passed to the desired library function, *args*, and the address of the desired library function, *&libfn* are placed on the stack.

Figure 1c shows the contents of the stack after the current function completes its execution and returns. Since the old return address was overwritten with *&VCS*, the VCS is now executing. The *locals* region for stack frame 0 was popped off of the stack when the previous function completed its execution, so the top of the stack contains the arguments for the desired library function and the address of the library function, as shown in Figure 1c. At this point, it is the job of the VCS to move the arguments from the stack to the argument registers, and move *&libfn* to the PC.

3.1 Vulnerable Code Sequence (VCS)

The Ret2ZP attack depends on the existence of an appropriate VCS in the system. A VCS, is a consecutive sequence of instructions which are resident in the memory of the victim machine. It would be possible for a vulnerable code sequence to be composed of multiple discontinuous code sequences which are connected by intervening jump instructions but we ignore this possibility for the sake of simplicity, and because it is unlikely to occur in practice. Any valid VCS must satisfy the following set of constraints that our tool checks for.

1. The final instruction in the sequence must copy data from the stack to the PC register. The execution of this final instruction which transfers control to the desired library function.

2. The sequence must contain no instruction which writes data to the PC register, other than the final instruction. This constraint ensures that the instruction sequence is continuous.

3. The sequence must move data from the stack into some subset of the argument registers.

4. The sequence must not write data to the stack. This is required because any data written to the stack might overwrite the library function arguments or the library function address which are already on the stack.

Figure 2 shows an example of a VCS. The first line of code loads argument registers r0 and r1 from the stack which is addressed by the stack pointer, SP. The final line of code executes a function return by popping the top of the stack and placing the popped value in the program counter, PC. Notice that the stack contents can be read using a load-type of instruction such as ldm, or a pop instruction which also updates the stack pointer.

4 VCS Detection

We have implemented a tool which identifies the presence of vulnerable code sequences in an executable. Detection of these sequences will allow the software to be patched, removing the vulnerability before the software is deployed. In order to analyze the executable, our tool uses the Radare toolkit [1] to disassemble the executable and generate assembly code which we can process. Once the executable is disassembled, our tool performs a single-pass scan of each instruction to identify any VCS.

An important aspect of a VCS is that it must move data from the stack into the argument registers. To

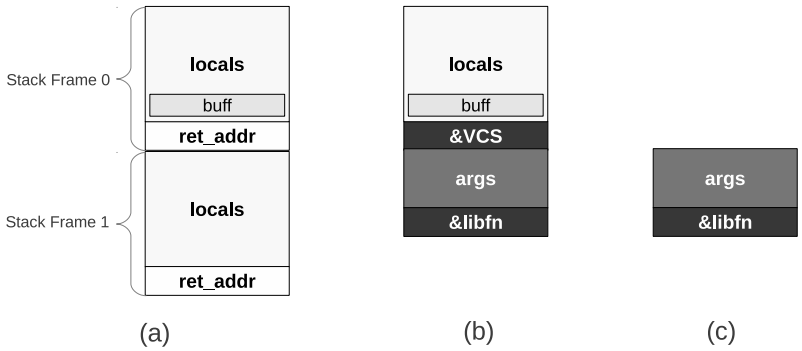

Figure 1: Stack contents during Ret2ZP attack, (a) before buffer overflow, (b) after buffer overflow, (c)start of VCS execution

```
ldm sp, {r0, r1}
add sp, sp, #8
pop {pc}
```

Figure 2: Vulnerable code sequence example

assist our discussion of this topic, we refer to a register as being *stack-controllable* at a point during program execution if the current value of the register was copied directly from the stack. A register becomes stack-controllable when an instruction loads stack contents into the register, and it is no longer controllable after the registers value is modified in any way, other than loading it from the stack.

The executable is scanned linearly from beginning to end, and at each line a set of stack-controllable argument registers R is maintained. A VCS is detected if an instruction is encountered which loads the PC from the stack, and $R \neq \emptyset$. Additional constraints are that the VCS should contain no branches to any address not taken from the stack, and that no instruction within the VCS should write to the stack. To enforce these constraints we set $R = \emptyset$ whenever either a non-stack branch or a write to the stack are encountered.

Figure 3 contains the pseudocode describing our VCS detection algorithm. In the pseudocode we use the following definitions:

- A is the set of argument registers.
- PC refers to the program counter register.
- R is the current set of stack-controllable argument registers.
- I is the ordered sequence of instructions in the executable.

1. $R = \emptyset$
2. foreach $i \in I$
3. foreach $r \in A$
4. if $i \in L_r$ then $R = R \cup r$
5. if $i \in M_r$ then $R = R - r$
6. if $(i \in W)$ OR $(i \in B)$ then $R = \emptyset$
7. if $i \in L_{PC}$ AND $R \neq \emptyset$ then
8. print VCS Detected
9. $R = \emptyset$

Figure 3: VCS Detection Pseudocode

- L_r is the set of instructions in the executable which load register r from the stack.
- M_r is the set of instructions in the executable which modify register r without loading register r from the stack.
- W is the set of instructions in the executable which write data to the stack.
- B is the set of instructions in the executable which branch to an address not taken from the stack.

The algorithm in Figure 3 initializes the set of stack-controllable registers on line 1, and enters a loop starting on line 2 which iteratively processes each instruction of the executable. The set of stack-controllable registers is updated on lines 3 - 5 where an argument register is added to the set if it is loaded from the stack, and deleted from the set if it is modified. A check for non-stack branch instructions and stack writing instructions is performed on line 6. The end of a VCS is detected at line 7 as an instruction which moves the stack contents to the PC while

File Name	C	exec.	# VCS
aes_expanded_key	508	28878	1
aes_set_key	481	27449	1
Bitband	515	25151	0
boot_demo1	491	32923	0
boot_demo2	510	34313	0
boot_demo_eth	529	67360	0
enet_io	1234	66532	0
enet_lwip	741	66627	0
enet_ptpd	1095	66621	1
enet_uip	690	62218	0
hello	539	36919	0
gpio_jtag	543	62218	0
Graphics	719	24014	0
interrupts	616	29175	0
mpu_fault	563	31472	0
Pwmgen	470	24623	1
qs_ek-lm3s6965	5178	66726	0
sd_card	731	62332	0
timers	497	26462	0
uart_echo	515	30110	0
Watchdog	476	25338	0

Table 1: VCS Detection Results

$R \neq \emptyset$. If a VCS is detected then success is announced (line 8) and the stack-controllable set is reset to start scanning for a new VCS in the remainder of the executable (line 9).

5 Experimental Results

We used our vulnerability analysis tool to detect vulnerable code segments in a set of same ARM executables. As our benchmark set of ARM executables we used the example programs provided for use with the EK-LM3S6965 Evaluation Board from Texas Instruments [2]. The board uses an ARM-based TI Stellaris LM3S6965 microprocessor. The sample programs are provided in C which we compiled for the LM3S6965 processor using the GCC compiler [18].

Table 1 shows the results of our analysis of all of the sample ARM-executables. Each row contains the results for a different program and the first column contains the name of the program. The second and third columns show the size of the original C programs and the size of the compiled executable. The final column labeled **# VCS** shows the number of vulnerable code segments found in each executable. The table shows that almost 20% of these executables contained a vulnerability to the Ret2ZP attack. The total CPU time required to detect VCS in all examples is 0.117 seconds.

```
ldmibeq sp!, {r0, r1, r2, r4, r5, r6,
              r7, fp, sp, pc}
```

Figure 4: VCS example in Pwmgen

The vulnerable code sequences detected in these examples were all comprised of a single line of code which loaded the argument registers as well as the PC. An example is shown in Figure 4 which was found in the Pwmgen sample program.

6 Conclusions

We have developed a tool which is used to detect Ret2ZP vulnerabilities in ARM executables. The popularity of return-oriented programming attacks underscores the need for the tool that we have developed. Our experimental results show that almost 20% of the programs that we evaluated actually contained vulnerabilities. Since the frequency of vulnerabilities can be expected to increase with program size, the detection of vulnerabilities becomes even more important for larger systems.

References

[1] **radare**, reverse engineering toolkit. http://radare.org.

[2] Texas Instruments EK3S6965 Evaluation Kit. http://www.ti.com/tool/eki-lm3s6965.

[3] M. Abadi, M. Budiu, U. Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[4] Aleph One. Smashing The Stack For Fun And Profit, 1995. http://insecure.org/stf/smashstack.html.

[5] Divya Arora, Srivaths Ravi, Anand Raghunathan, and N.K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 178–183. IEEE Computer Society, 2005.

[6] Itzhak(Zuk) Avraham. Non-Executable Stack ARM Exploitation Research Pa-

per. In *BlackHat Security Convention*, 2011. https://media.blackhat.com/bh-dc-11/Avraham/BlackHat_DC_2011_Avraham_ARM Exploitation-wp.2.0.pdf.

[7] Mihai Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51. ACM, 2006.

[8] cOntext. Bypassing Non-Executable Stack During Exploitation Using Return-to-LibC. http://infosecwriters.com/text_resources/pdf/return-tolibc.pdf.

[9] M.L. Corliss, E.C. Lewis, and A. Roth. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News*, 33(1):65–72, 2005.

[10] Ang Cui and Sal Stolfo. Print Me If You Dare Firmware Update Attack and the Rise of Printer Malware, 2011. http://ids.cs.columbia.edu/sites/default/files/CuiPrintMeIfYouDare.pdf.

[11] Solar Designer. Getting around non-executable stack (and fix), 1997. http://seclists.org/bugtraq/1997/Aug/63.

[12] Ú. Erlingsson, M. Abadi, Michael Vrable, M. Budiu, and G.C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[13] Nicolas Falliere. Stuxnet Introduces the First Known Rootkit for Industrial Control Systems, 2010. http://www.symantec.com/connect/blogs/stuxnet-introduces-first-known-rootkit-scada-devices.

[14] Aurélien Francillon, Daniele Perito, Claude Castelluccia, and Inria Rhône-alpes. Defending Embedded Systems Against Control Flow Attacks Categories and Subject Descriptors. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26, 2009.

[15] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory violations. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 652–657. IEEE, 2009.

[16] Koji Inoue. Energy-security tradeoff in a secure cache architecture against buffer overflow attacks. *ACM SIGARCH Computer Architecture News*, 33(1):81–89, 2005.

[17] ARM Ltd. Procedure Call Standard for the ARM Architecture, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf.

[18] GCC the GNU compiler collection. gcc.gnu.org.

[19] Milena Milenković, A. Milenković, and E. Jovanov. Hardware support for code integrity in embedded processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 55–65. ACM, 2005.

[20] Mario Morales and Michael J Palma. Intelligent systems : The next big opportunity, 2010. www.idc.com.

[21] G. Novark, E.D. Berger, and B.G. Zorn. Exterminator: automatically correcting memory errors with high probability. *Communications of the ACM - Surviving the data deluge*, 42(6):1–11, 2008.

[22] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, 2010.

[23] R.G. Ragel, S. Parameswaran, and S.M. Kia. Micro embedded monitoring for security in application specific instruction-set processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 304–314, 2005.

[24] 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 Communications and Security*, pages 552–561, 2007.

[25] T. Zhang, X. Zhuang, S. Pande, and Wenke Lee. Anomalous path detection with hardware support. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 43–54, 2005.

ABOUT THE AUTHORS

Zi-Shun Huang - *zishunh@uci.edu*

Ian G. Harris - *harris@ics.uci.edu*

Center for Embedded Computer Systems

University of California Irvine
