Reprogramming embedded systems using Return Oriented Programming

Sam Mitchell and Nathanael Weidler
Deptartment of Electrical and Computer Engineering
Utah Stat University
Logan, Utah 84322
e-mail: samuel.alan.mitchell@gmail.com, NWeidler@gmail.com

***Abstract*—This paper describes the theory and implementation of a return-oriented programming attack on a ARM-based device. The attack reprograms the device to execute our desired code upon reset. We describe the gadget searching process and the code injection method.**

***Index Terms*—Return Oriented Programming, Security, ARM, Harvard architecture.**

I. Introduction

Single-purpose embedded devices such as voting machines or vehicle guidance controllers are generally considered to be secure machines. Previous work has shown that Return Oriented Programming (ROP) methods can be used to attack ARM-based devices [1] [2].

ROP is a type of buffer-overflow attack that modifies the return address of the existing program, causing the program to execute existing code. In x86 architectures, ROP attacks generally jump into libc to control the behavior of the compromised program. An attack on an ARM-based device uses similar techniques — manufacturer-provided peripheral driver libraries provide sufficient code-space to implement a devastating ROP attack.

A. Structure of paper

The organization is as follows: in Section II, the system design and modifications required to enable the ROP attack are presented. Section III contains the desired program and code required to inject the program into the system without ROP. Gadgets to utilize in the ROP attack and the required sequence of execution are proposed in Section IV. Section V presents the implementation methods and results of the attack. Conclusions are discussed in Section VI.

II. System design

A traditional ROP attack is performed using strcpy() or UARTgets() buffer overflow. This adds complexity to the problem when trying to transmit a backspace (0x08) character across the line, because the buffer treats it as a backspace. The UARTgets() function was altered to still insert the backspace character.

Another protection built in with the function UARTgets() is the expected buffer size limit. The function normally accepts the buffer size as an argument, which would prevent buffer overflow, thereby rendering any ROP impossible. This functionality was also disabled.

As part of the preparation for the ROP implementation, we disabled some optimizations in order to simplify the attack. The compiler flag -O0 was used instead of -O2, which made the assembly code more readable. Another compiler flag, –no_protect_stack, was used to remove canaries which alert the program when the stack is corrupted. The linker flag, –no_remove, ensured that the included libraries were flashed to the board even if the code wasn't executed. This doubled our available code space, which allowed for more precise selection of gadgets.

III. Reprogramming method

The target of this project is to insert a program (see Figure 1) that will run at reset. The current main function is located at memory address 0x2aca. Executable memory must be reprogrammed through the Flash module (located at 0x400fd000). There are multiple methods to reprogram the code space at 0x2aca.

```
Start
        add R0, #0x1        ; 0xF1000001
        b Start             ; 0xE7FC
```

Figure 1: The program to be inserted.

Flash memory can either be erased or programmed. Memory is erased (the bits are cleared to a value of 1) in 1kb chunks. Programming can only bring a bit from high to low (1 to 0). The most simple method to insert the program at main would be to overwrite existing code currently located at main. This method is only available if the desired command has 1s located in the same position as the existing command's 1s. See Figure 2. The target program does not have convenient programming

```
Current instruction      0xE92D4FF0
Desired instruction      0xF1000001
```

Figure 2: Writing the desired instruction doesn't work because the current instruction bits would require 0 to 1 programming.

instructions, as can be seen in Figure 2. The memory at main must first be erased (written to 1s) then programmed. The procedures to erase, reprogram, and an alternate reprogramming sequence are located in Figures 3, 4, 5, respectively.

```
// base address
uint32_t * FLASH = (uint32_t *) 0x400FD000;
FLASH[0x0] = 0x2800; // address to erase
// clear the area 0x2800-0x2C00
// perform erase command
FLASH[0x8/4] = 0xA4420002;
```

Figure 3: Erasing sequence.

```
// address to program
FLASH[0x0] = 0x2ACA;
// add instruction
FLASH[0x4/4] = 0xF1000001;
// perform write command
FLASH[0x8/4] = 0xA4420001;

// address to program
FLASH[0x0] = 0x2ACE;
// branch instruction
FLASH[0x4/4] = 0xE7FC0000;
// perform write command
FLASH[0x8/4] = 0xA4420001;
```

Figure 4: Reprogramming sequence.

```
// address to program
FLASH[0x0] = 0x2ACA;
// add instruction
FLASH[0x100/4] = 0xF1000001;
// branch instruction
FLASH[0x104/4] = 0xE7FC0000;
// perform write command
FLASH[0x20/4] = 0xA4420001;
```

Figure 5: Alternative reprogramming sequence.

Translating the rewrite procedure into assembly will require a load / move / pop instruction to populate registers, followed by a store command to write to memory. The load / move / pop command is discussed in Section IV.

IV. Gadgets

The basic operation of ROP is performed by redirecting the locations jumped to via buffer overflow. This method does not actually insert any executable code onto the stack — existing code is merely utilized in creative ways. Gadgets are the building blocks of ROP. Any gadget used in ROP must contain a return-like command.

Because Thumb assembly does not contain any return commands, alternatives to this command must be used. Two such instructions are bx and pop {pc}. The bx / pop and the preceding lines of code are what constitute a gadget. Many gadgets exist, but careful selection can produce a turing-complete instruction set.

The flash rewrite sequence contained in Figures 3 and 5 requires two operations: load and store. Our search for gadgets resulted in two effective sets of instructions, located in Figure 6.

```
; Gadget A
str R0, [R4,#0x0]
pop {R4,PC}

; Gadget B
mov R0, R4
pop {R4,PC}
```

Figure 6: Gadgets that provide the required load and store operations.

Gadget A: this gadget provides the ability to store data from R0 into the address specified at R4. It also causes the program to jump to the next instruction, while filling R4 with more data from the stack. This gadget is effective because R4 is constantly updated.

Gadget B: this gadget transfers the data from R4 into R0. There were no gadgets that would load R0 directly, so this method was a sufficient substitute. The data from the stack is transferred from the stack to R4 during Gadget A, followed by Gadget B where the data is shuttled to R0 while R4 is repopulated. Finally, the data is stored into the desired location via Gadget A.

The design of the ROP was taken directly from the code in Figures 3 and 5. The gadgets in Figure 6 were combined in a pipelined fashion in order to minimize operations. Our implementation was still rather bulky at 13 required returns. The number of returns could have been reduced by finding gadgets containing the STR2 command, which stores two words at an address. Table I describes the order that the gadgets should be executed in order to rewrite the flash memory of the TM4C123GH6PM.

V. Implementation and results

The ROP attack was first approached by determining the size and boundaries of the stack (see Figure 7). Once the boundaries were determined, we replaced the location which would be popped into the program counter (PC) with the address of Gadget A. Each successive call (shown in Table I) was determined by overwriting the values to be placed into the R4 and PC registers.

The attack outlined in Table I was performed and resulted in the desired functionality until the device was reset. Upon reset, the program would jump to a scatter function located

Table I: The required inputs and gadget sequence to execute the flash rewrite sequence in Figures 3 and 5.

Gadget	R4 input	Resulting code
A	0x00004800	*// Erase procedure*
B	0x400FD000	uint32_t * FLASH = (uint32_t *) 0x400FD000;
A	0xA4420002	FLASH[0x0] = 0x4800;
B	0x400FD008	FLASH[0x8/4] = 0xA4420002;
A	0x00004BD8	*// Write procedure*
B	0x400FD000	FLASH[0x0] = 0x4BD8;
A	0xF1000001	FLASH[0x100/4] = 0xF1000001;
B	0x400FD100	
A	0xE7FC****	FLASH[0x104/4] = 0xE7FC****;
B	0x400FD020	
A	0xA4420001	FLASH[0x20/4] = 0xA4420001;
B	0x400FD008	
A		*// Return to 0x4BD8*

in the previously erased region. This resulted in an interrupt which prevented the device from executing the program located at the address of main(). This was solved by inserting data falsification at the scatter function.

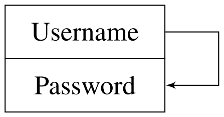

Figure 7: The ROP attack construction.

VI. Conclusion

This paper demonstrates an ROP attack on a Harvard-architecture device. It is shown that it is possible to insert a program that will return to execution after restart, while the device is still running. Future research will address the attack of the device without limitations imposed in Section II.

References

[1] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage," *Proceedings of EVT/WOTE*, vol. 2009, 2009.
[2] T. Kunz, "Rop on the arm," February 2014.