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 implementation of high-throughput password cracking devices. We consider architecture-aware implementations of password crackers on FPGA and X86 architectures. An analysis of speed and cost efficacy is included. This paper describes the theory and implementation of a return-oriented programming attack on a TM4C123GH6PM microcontroller. We describe the gadget searching process, as well as the 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 clear and reset Harvard architecture-devices.

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. Harvard architectures use 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 are proposed in Section IV. The design of the ROP attack are outlined in Section V. Section VI presents the implementation methods and results of the attack. Conclusions are discussed in Section VII.

II. System design

III. Required assembly

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.

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 will be discussed in Section IV.

```
// 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[0x8/4] = 0xA4420001;
```

Figure 5: Alternative reprogramming sequence.

IV. Gadgets

V. ROP design

VI. Implementation and results

VII. Conclusion

This paper considers the efficient computation of passwords. Multiple methods to increase hashing throughput are presented. It is shown that hardware implementation of a password cracker provides more throughput per unit dollar than a software implementation. Future research will address the efficacy of different architectures in password computation.

References