Return Oriented Programming on an Embedded System Using Code Injection

Justin Cox and Tyler Travis
Deptartment of Electrical and Computer Engineering
Utah State University
Logan, Utah 84322
email: justin.n.cox@gmail.com, tyler.travis@aggiemail.usu.edu

***Abstract*— This paper describes the process of implementing a code injection attack on an ARM Cortex M4 through Return Oriented Programming. The paper will describe the process of selecting useful snippets of code found in the ROM of a microcontroller. The paper will end by showing the injected program working on the microcontroller.**

***Index Terms*—Return Oriented Programming (ROP), stack overflow, security, embedded system, canary.**

I. INTRODUCTION

Every year, the amount of devices with embedded systems greatly increases. With the demand that every device connect to the internet (Internet of Things), in the near future almost all devices will have an embedded system. That being said, generally these embedded devices are not manufactured with security as the top priority.

Buffer overflow attacks have been widely demonstrated and used to allow a hacker to gain control over a system or program. Most of the buffer overflow techniques on x86-64 architectures have been well researched and defended against. However, this is not the case for embedded systems. Using more advanced buffer overflow techniques such as Return Oriented Programming (ROP), an attacker is able to utilize the manufacture's driver libraries and boot code found on the ROM to take over an embedded device. This vulnerability will become more threatening when most common, household items have embedded devices such as microcontrollers.

II. VULNERABILITY OVERVIEW

A. Overview of ROP

As stack based, buffer overflow attacks became more common, programmers and researchers came up with methods to defend against these attacks. A simple method of defense is to prohibit a section of memory to be both wrote to and executed from. If an attacker was able to accomplish a buffer overflow, the attacker would not be able to return and execute the malicious code on the stack. In response to this defense, attackers came up with ways to use the code already found on most systems, e.g. libraries such as libc. One of these more advanced methods of attack is Return Oriented Programming (ROP).

ROP allows the attacker to gain control flow over a program by using useful snippets of code found in drivers and/or libraries called "gadgets". After execution, each gadget needs to end with a return command. The return command allows the attacker to return to another gadget or eventually to a malicious program.

In the case of an ARM architecture, there are no "return" commands so branches and pop instructions can be used instead to achieve the same result [1].

B. Buffer Overflow

The vulnerable function that will be exploited to allow a buffer overflow is the following:

```
void echo(void){

  char buffer[32];

  UARTgets(buffer, sizeof(buffer));

  UARTprintf("%s \n", buffer);

}
```

The UARTgets() and UARTprintf() functions are provided from the TivaWare libraries from TI. In order to be able to overflow the character buffer of size 32, we had to remove the bounds checking done in the UARTgets() function. It is worth noting that for certain user-input obtaining functions, the function stops getting the input when it encounters the null character 0x00. This is not the case for UARTgets(), making the ROP attack easier to develop. The malicious code will be inserted onto the stack as the character buffer is overflowed.

C. Overcoming Canary Stack Protection

Another defense used to stop buffer overflow attacks is the use of a canary. The main objective of a buffer overflow is to overwrite the functions return address. A canary is placed in between the buffer and the return address. Before the program returns from the return address, it checks to see if the canary has been modified. If it has, it throws an exception and program execution is halted.

The IDE being used to program the echo() program onto the microcontroller is Keil uVision. In Keil, there are options available to turn on stack protection. For stack protection in Keil, the canaries are defined by the programmers. A randomly generated canary is the best choice for increased security. However since the generation of random canaries

takes a toll on the overall performance of the microcontroller, often times a static canary is used. A common canary that is used is a 32-bit word containing nulls, e.g. 0x00000000 [2]. The authors chose to use the null canary and a later section in this paper will explain how the canary was overcome.

III. CODE INJECTION

In order to inject the attackers code, the microcontroller's flash memory needs to be erased and then reprogrammed. The microcontroller that will be used is the TM4C123G, more specifically, the Tiva C LaunchPad from Texas Instruments. The TM4C123G has a ARM Cortex M4 architecture that uses the Thumb-2 instruction set.

In order to erase the flash, the following must be performed:

1) Write the flash key and MERASE bit to the FMC register
2) Wait for flash erase routine to finish

In order to reprogram the flash, the following must be performed:

1) Write data to the FWBn registers
2) Write the start address of program into the FMA register
3) Write the flash key and WRBUF bit to the FMC2 register
4) Wait for flash write routine to finish

In order to accomplish the above procedures, the ROP gadgets will need to be able to perform loads from the stack as well as perform a store to a desired memory location.

A program was written in C++ to send the malicious data over a COM port that interfaces with a UART module on the microcontroller.

IV. GADGET SELECTION

There is one function that is needed to erase the flash, write to the flash buffer, and commit the flash write buffer into the flash memory. That function needs to write certain data to an in-memory register. This overall fuction is broken down into 3 subsections:

1) Load the correct data off of the stack into a register
2) Load the correct address off of the stack into a register
3) Store the data into the address

Gadget one through three correspond to the list above.

Gadget one:

```
ldr r0, [r4, #4]
pop {r4, pc}
```

Gadget two:

```
pop {r4, pc}
```

Gadget three:

```
str r0, [r4, #0]
pop {r4, pc}
```

Gadget four is used for a delay when the flash is getting erased or written to

Gadget four:

```
tst.w r0, #8
ite ne
movne #15
moveq #14
pop {pc}
```

The actual instruction functionality of gadget four isn't important, only the amount of time it takes to execute.

V. IMPLEMENTATION

Using the gadgets explained in the previous section, a carefully crafted byte string was sent over UART into the character buffer on the microcontroller. Assuming the attacker has the ability to attempt the buffer overflow many times, the canary value was discovered to be a 32-bit word of null characters. This null canary was precisely placed in the byte string to preserve the canary.

Following the canary, the initial gadget was placed in the spot of the former return address. This initial portion of the attack byte string is illustrated in Figure 1.

Fig. 1. An example of how the buffer overflow attack is initialized.

The next part of the attack string handles the flash memory erase routine. The section of stack memory pertaining to the flash erase routine is further explained in Figure 2.

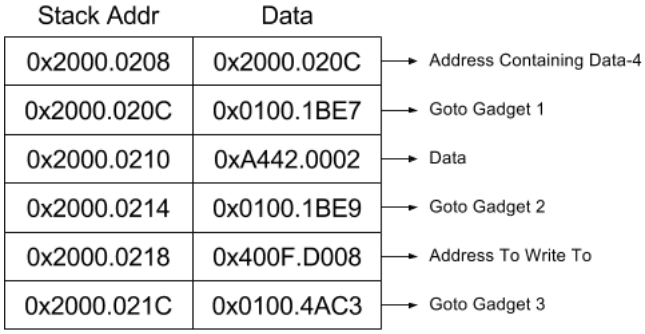

Fig. 2. An example of how the gadgets are stored on the stack to perform multiple load and stores.

After the flash erase routine is performed, the attack byte string has a large number of delay gadgets to allow the erase to finish. A large amount of delay gadgets are also used when the flash write routine is performed.

The next part of the attack string is used to load the new program data into the flash memory write-data buffers. This routine is repeated for all of the opcodes that need to be loaded.

The last portion of the attack sets the PC address to the beginning address of the newly loaded program. The program will begin without having to restart the microcontroller. This feature allows an attacker to inject the malicious code remotely.

VI. RESULTS

The authors were able to successfully perform a code injection attack using ROP. The microcontroller was initially programmed to perform an "echo" functionality. Using a buffer overflow with ROP, the authors were able to erase the flash, reprogram the flash, and start the new program. The new program flashes an LED found on the Tiva C Launchpad at a human observable rate. Here is a picture of the successful hack.

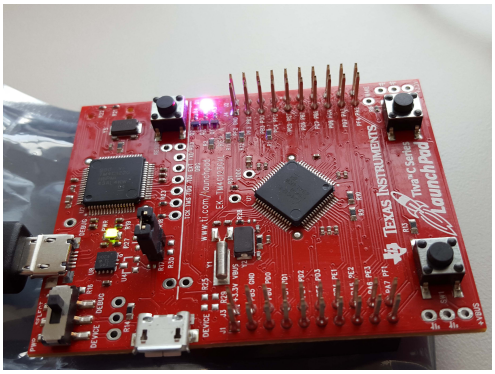

Fig. 3. Successful code injection of a blinky program.

A. Problems Encountered

The flash erase and flash write routines require a delay time. In order to satisfy the delay time, many delay gadgets were used to "waste time". It would be better to find a gadget that is able to poll the READY bit. This is something that can be enhanced in a future implementation. It would allow the attack byte string to be much smaller.

VII. CONCLUSION

This attack proves that code can be injected onto the TM4C123G microcontroller. Given the right circumstances, code can be erased and executed by exploiting the internal ROM by using ROP on an ARM device. This implies that all TM4C123G microcontroller of this family are vulnerable to this type of attack. Microcontrollers and embedded systems need to achieve better security to keep up with newer attacks to their systems.

REFERENCES

[1] L. Davi, A. Dmitrienko, A. R. Sadeghi, M. Winandy, "Return-Oriented Programming without Returns on ARM," System Security Lab, Ruhr University Bochum, Germany, Tech. Rep. HGI-TR-2010-002, July 2010.

[2] G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection," Core Security Technologies, April 2002.

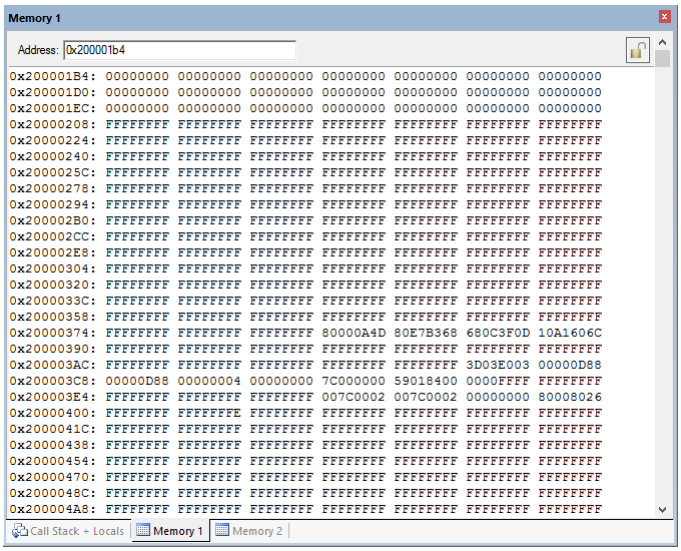

Fig. 4. The stack memory before the attack.

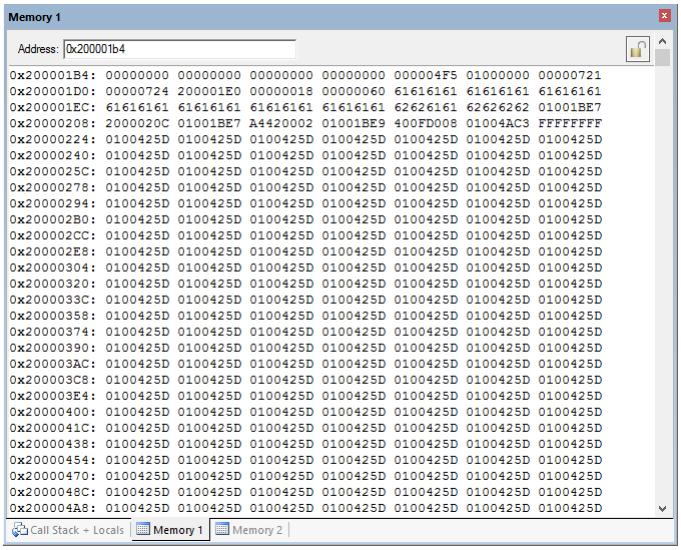

Fig. 5. A beginning portion of the stack memory after the attack.

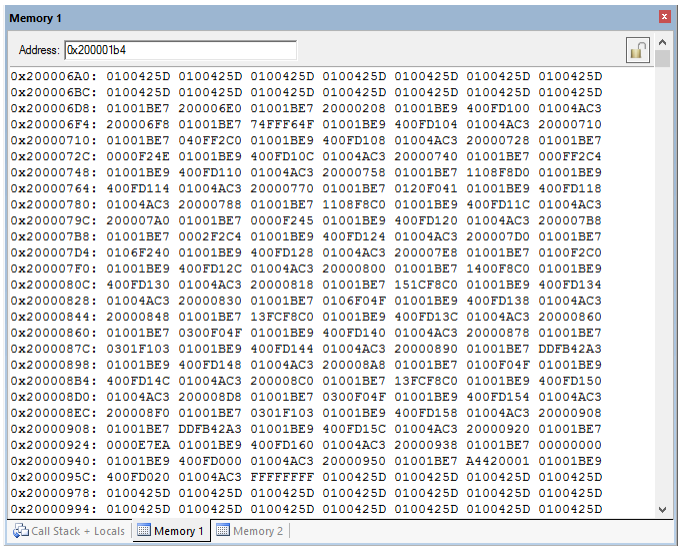

Fig. 6. A portion of the stack memory with the opcodes after the attack.