CSE 608 Fall 2013
Project 2 Report

Vinay Krishnamurthy, Amogh Akshintala

October 6, 2013

Introduction

This project was an empirical study on ROP based attacks on Android platform, specifically on ARM platforms. We also compare ROP on ARM vs ROP on Intel (x86) platforms.

What is ROP?

Return Oriented Programming (ROP) is an exploit technique that lets an attacker run malicious code, in spite of having security mechanisms such as DEP (Data Execution Prevention) or NX (Never eXecute)[5].The attacker need not inject any malicious code, such as shellcode, but he/she can build this code from existing instruction sequences from shared libraries. The only constraint is that the code sequences that the attacker builds will always end with a *ret* instruction.

ROP on x86 platforms

We have already gone through a paper in class [1], which talks about ROP attacks on both CISC and RISC architectures. ROP is always the second step while mounting an attack, which follows buffer overflow (overflow on either stack / heap). The general idea of ROP is to build gadgets, a sequence of instructions that end with *ret*. These instructions are borrowed from various existing libraries such as *libc*. Usually these gadgets are chained together to get the intended malicious behavior.

Additionally, since x86 instruction set has no fixed demarcation between successive instructions, it is possible to grab part of an instruction, and make it behave like a different instruction altogether. x86 calling convention requires the return address to be pushed onto the stack. After a stack overflow, the attacker can supply the return address as the address of the first ROP gadget. This is a little bit different from that on ARM, which is discussed in detail below.

ARM instruction set

ARM instruction set is a 32 bit RISC based instruction set. It has 16 general purpose registers, with registers R13, R14 and R15 as the Stack Pointer register, Link Register and Program counter respectively. Link register stores the return address when a function is called. Note that there are no `call` or `ret` instructions in ARM, but they are implemented using branch instructions such as `bx`. ARM opcodes are 32-bit opcodes, and are word aligned. THUMB opcodes, however, are half word aligned. ARM instructions can belong to any one of the following 4 categories [4]:

- Arithmetic instructions
- Logical instructions
- Comparison instructions
- SIMD (Single Instruction Multiple Data) instructions

ROP attack on ARM

As discussed previously, while mounting an ROP attack on x86, the attacker would have to change the return address on the stack to the address of the first ROP gadget. On ARM architecture, the return address will be stored in the Link Register (`lr`). This brings in the first challenge, and overflowing the stack will not suffice. Here's a snippet of code that we wrote to check if the buffer overflow would cause the app to crash.

```
void overwriteBuffer(const char *str)
{
        __android_log_print(ANDROID_LOG_ERROR, DEBUG_TAG, "Return address = %x",
          (unsigned int) __builtin_return_address(0));
        char overflowBuffer[5];
        int i;
        for (i = 0 ; i < strlen(str); i++)
          overflowBuffer[i] = str[i];
        __android_log_print(ANDROID_LOG_ERROR, DEBUG_TAG, "Return address = %x",
          (unsigned int) __builtin_return_address(0));
}

jstring
Java_com_example_proj2_MainActivity_overflowBuf(JNIEnv * env, jobject this)
{
        char *attackStr = (char *)malloc(1024);
        int i;
        for (i = 0; i < 1024; i++)
          attackStr[i] = 'A';
        attackStr[i] = 0;
        overwriteBuffer(attackStr);
}
```

Figure 1: Sample overflow program written in C, and exported to Java using JNI

In this code, we're trying to overflow the buffer in `overflowBuffer` method. After building this code as a shared library using Android NDK, we ran the app on an emulator and on a real hardware device. However, in both the instances, the app did not crash, or cause any unintended behavior. This was because the return address was intact, and this was confirmed by logging the return address before and after the overflow.

We tried a heap corruption attack as well, by malloc'ing lesser number of bytes for `attackStr` string. For our tests, we tried allocating just 512 bytes for this string. The app did crash this time, but heap corruption was detected by *libc*, which ended the program execution. Heap corruption is something that we chose not to follow at the moment, because overflow in heap would only be fruitful if there is critical information or a function pointer on the heap.

Additionally, there is no one `ret` instruction in ARM ABI. In ARM instruction set, a `ret` is performed by `pop lr`, followed by `bx lr` instruction sequence. Note that the `pop` instruction might pop more registers than just `lr` register. Here's a snippet from *memcpy* function from Bionic *libc*, that shows the function return.

```
1d078:        e8bd4011        pop {r0, r4, lr}
1d07c:        e12fff1e        bx lr
```

Figure 2: Return in ARM ABI

We also went over the bionic *libc* disassembled code to check for instructions that could be used to spawn a shell. The shellcode on ARM as documented in [3] is given below:

```
e28f3001        add r3, pc, #1        ; 0x1
e12fff13        bx r3
4678            mov r0, pc
300c            adds r0, #12
46c0            mov r8, r8
9001            str r0, [sp, #4]
1a49            subs r1, r1, r1
1a92            subs r2, r2, r2
270b            movs r7, #11
df01            svc 1
622f            str r7, [r5, #32]
6e69            ldr 1, [r5, #100]
732f            strb r7, [r5, #12]
0068            lsls r0, r5, #1
```

Figure 3: ARM shellcode

We were not able to instruction sequences for the shellcode that had subsequent `pop` and `bx` instructions on `lr` register. We were able to locate a variant of the first instruction: `add r3, pc, r2`. This instruction could be used to simulate the first instruction of the shellcode, by moving a 1 to `r2` register (which would be the first gadget) But, this is all we could find in *libc* library, and we could not locate any subsequent `pop` and `bx` instructions.

The other major difference between generating ROP gadgets for x86 and ARM is that there are no unintended instruction sequences in ARM. Every instruction is either 2 bytes (ARM) or 4 bytes (THUMB) long. Hence, it is not possible to jump to the middle of an instruction stream and look for a branch instruction. It is possible to get such unintended sequences on x86 architecture, though.

Conclusion

In this report, we've tried pointing out the differences between mounting an ROP attack on x86 and on ARM. As we can see, performing an attack on ARM is definitely more complex than on x86 platform. That being said, it is not entirely impossible to perform one. Our study was mainly restricted to *libc* library's instruction sequences. However, attackers will have access to many libraries, and they could write their own ROP gadget generators to build a chain of ROP gadgets.

Will ASLR mitigate this problem?

ASLR cannot prevent an ROP attack from happening because the attacker would build the ROP gadgets offline, and pass the gadgets as payload to the phone. Enabling ASLR will not protect against anything as the attacker is not relying on *libc* present on the device, to extract the instructions.

Android OS, today, is still a 32-bit OS, so it is possible to mount an attack as discussed in [2] theoretically, but in practice, it will catch the attention of the user. Let's take an example, say there is an app *App1*, which is being targeted by an attacker to break the ASLR. The attacker cannot attempt brute force because, the user will incessantly get the '*App1 has stopped working unexpectedly*' message.

References

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. Manuscript, 2009. Online: `https://cseweb.ucsd.edu/ hovav/papers/rbss09.html`.
[2] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. *In Proceedings of the ACM Conference on Computer and Communications Security, 2004.*
[3] Shellcode database on shell-storm website. `http://www.shell-storm.org/shellcode/files/shellcode-696.php`.
[4] Gaurav Kumar, Aditya Gupta. A Short Guide on ARM Exploitation.
[5] Wikipedia article on ROP. `http://en.wikipedia.org/wiki/Return-oriented_programming`.