Return Object Programming Gadget Utility for ARM using Capstone Engine and C

Theodore Kim
The Art of Binary Exploitation: Embedded Systems, Spring 2020
New York University Tandon School of Engineering

***Abstract*—Return Object Programming is an exploitation methodology which allows for arbitrary code execution through a stack overflow vulnerability and manipulation of the call stack. The challenge of performing a ROP exploit is finding code segments to jump to execute arbitrary code actions. This report outlines an implementation of a ROP gadget utility using the Capstone Disassembly Engine API in C. The utility searches binaries for return instructions and counts backwards to find useful gadgets.**

I. Introduction

Return Object Programming (a.k.a. ROP) is a methodology of binary exploitation which allows for arbitrary code execution through successive return calls to arbitrary locations in the call stack. ROP exploits are another means by which to exploit buffer overflow vulnerabilities and use a binaries' own code to execute actions not originally intended by the application (such as spawning a shell process in an application that may not already use functions like "shell" or "execv").

The basic attack vector for a ROP exploit is as follows:

1. The tester discovers a buffer overflow vulnerability which allows them to overwrite the return address in a call stack. Often times ROP exploits require successive function returns and therefore necessitate several return addresses and data values to be written to the stack.

2. The tester finds a position in the source code which accomplishes a desired operation (such as loading a value from the stack into a specific register) followed closely by another return instruction (this is referred to as a gadget) to allow for a successive jump to another desired gadget.

3. Successive return jumps are made until enough operations are performed to reach the desired execution output and the program is exitted.

The primary advantage of a ROP exploit is that it is very flexible. using a ROP exploit and sufficiently complex source binary may allow for a complex assembly-level programming interface. Additionally, ROP exploits may also jump into dynamically loaded libraries (given that the mount address of the dynamic library is known) thus increases the available code base for ROP instructions. Furthermore, library ROP jumps are useful as many libraries (such as glibc) are open source and therefore can be examined by the attacker for ROP gadgets when the taregt application source is unknown.

ROP exploits have several defenses. The most common, the utilization of Address Space Layout Randomization (ASLR), prevents library addresses from being predict reliably and jumped into. Furthermore, stack canaries are an effective defense against ROP exploits as it prevents return calls from being made in the event of a stack corruption. While these defenses may be overcome with enough information leakage from the target binary, the most effective way to prevent a ROP exploit is to remove buffer overflows and the ability to overwrite the stack.

One of the difficulties of ROP exploits is finding gadgets to perform operations in the source code. The purpose of this project is to develop a utility to find ROP gadgets in an executable binary file and report them to the user, along with its address so that the address can be overwritten into the stack. Other utilities already exists which accomplishes this task like rp++ (https://github.com/0vercl0k/rp).

II. Design

The design of the system is actually simple. The basic algorithm of operation is as follows:

1. The ELF file is opened and its header is read to determine the endian order and the start of the code section
2. The code section is disassembled into its assembly instructions.
3. The instructions are looped through and all instructions that are equivalent to the C "return" function (i.e. a stack call frame exit) are found.
4. Each of the above return functions are stepped backwards for the designed gadget length and reported back to the user.

The primary challenge of the project is located return-like instructions. Unlike other architectures like x86, ARM does not have dedicated return opcode. Instead, there are multiple ways in which compilers interpret return. Therefore, the utility looks for any instruction that writes to the Program Counter (pc) register. Furthermore, a common return pattern for leaf function calls (i.e. functions that do not call other functions) is a jump to the contents of the link register (lr) which is where the branch and link instruction (bl) stores the return address before a function call. Therefore, this may be useful in accomplishing a return after a previous ROP call places a required return address to the lr register.

The last difficulty encountered during the design of the project was the two addressing mode of the ARM architecture: ARM (32-bit instructions) and THUMB (16-bit instructions). As ARM can dynamically change modes during execution, sequential instructions in an ARM binary may not disassemble consistent with their original intention as some byte sequences may yield different assembly instructions and operands depending on the current addressing more. Aside from predicting the addressing mode during static analysis, some ELF formats label code sections by their addressing mode. However, this is not consistent and may be disabled during compilation. The final design decision made was to allow the user to select their addressing mode as their would know, based on dynamic analysis, what addressing mode is being used in the arbitrary return context.

III. Implementation

The tool was developed in C. The project uses the Capstone Disassembly Engine to disassemble the raw binary into parsable assembly. The user interface is done through the command line. The Command Line Interface allows users to choose a desired ROP length and instruction length mode (THUMB or ARM).

IV. Testing

The utility was tested using GLibC (version 2.14) compiled for ARM. The following is an excerpt from the output of running the implemented utility:

```
[THUMB] 0xb45e2: movs r2, r0; bx lr;
[THUMB] 0xb45e6: nop ; bx lr;
[THUMB] 0xb45ea: nop ; bx lr;
[THUMB] 0xb45ee: nop ; bx lr;
[THUMB] 0xb4646: pop {r4, r5, r6, r7}; bx lr;
[THUMB] 0xafab2: bne #0xafaa4; bx lr;
[THUMB] 0xafadc: mov r0, r1; bx lr;
[THUMB] 0xafb82: pop {r4, r5}; bx lr;
[THUMB] 0xb49e8: add sp, #8; pop {r4, pc};
[THUMB] 0xb4a52: it eq; bxeq lr;
[THUMB] 0xb4a7c: mov.w r2, #0; mov pc, r3;
[THUMB] 0xb4c88: mov r0, r2; bx lr;
[THUMB] 0xb4c90: movne r0, #0; bx lr;
[THUMB] 0xb4c9c: lsr.w r0, r0, r2; bx lr;
[THUMB] 0xb4cc0: sub.w r1, r1, r3; bx lr;
```


Acknowledgement

This project utilizes the Capstone Disassembly Engine compiled for C. Done under the guidance of Stephen Ridley, for his course The Art of Binary Exploitation: Embedded Systems at New York University, Tandon School of Engineering, Spring 2020. The full source is available on my GitHub page: http://github.com/theo-kim/roparm.