A Compiler Pass to Detect ROP attacks

Akshay Utture (CS13B031)

Indian Institute of Technology Madras

ARTICLE HISTORY
Compiled November 14, 2017

ABSTRACT
Return Oriented Programming (ROP) attacks fall under the category of Buffer overflow attacks and are able to overcome the various buffer overflow counter-measures like canaries, making the stack non-executable, Address Space Layout Randomization (ASLR), etc. ROP attacks construct gadgets from pieces of code that end in a *ret* instruction, and stitch these together to construct their payload. As a part of this project I propose a compiler pass that will instrument some code at the end of every function, that will be able to detect ROP attacks and other Buffer Overflow Attacks, and terminate the program if it is detected.

KEYWORDS
Buffer overflow vulnerability, Return-oriented Programming, Compiler Pass

1. INTRODUCTION

Most malware work in 2 stages. The first involves subverting the normal execution of the code to execute something else. The second, is executing your payload, or the actual malicious code. One of the popular techniques used to subvert execution is the buffer overflow vulnerability, and even today, several inexperienced programmers unknowingly introduce this vulnerability into their program.

1.1. *Buffer overflow attacks*

Whenever a data is read into a buffer without checking if the bounds are being crossed, it is possible to give an input larger than the size of the buffer, and overwrite the addresses beyond the buffer. One popular way of using buffer overflow vulnerabilities is to overwrite the return address of the function with the buffer, and hence subvert execution.

Several techniques have been implemented to overcome buffer overflow vulnerabilities. One method is to use a canary near the return address and while returning from a function, ensure that the canary is not overwritten. Another method is to make the stack non-executable in order to prevent the attacker from adding the payload to the stack and executing it from there. Buffer overflow attacks have been able to circumvent these and many other proposed protection mechanisms. Address Space Layout Randomization (ASLR) was an effective technique used prevent these further attacks. ASLR randomized the code and data layout every time the program ran, so that the attacker would not know what address to overwrite in place of the original return address.

1.2. *Return-oriented Programming (ROP) attacks*

The Return-Oriented Programming (ROP) attack is a type of Buffer overflow attack which is able to overcome all these techniques. ROP attacks construct gadgets from pieces of code in the original program that end in a 'ret' instruction, and stitch these together to construct their payload. An ROP gadget is a sequence of instructions which do not have any branch, jump or return instructions, except for the last instruction, which should be a return instruction. After overflowing the buffer, the return address is overwritten and the next few bytes are replaced by the gadget addresses in the sequence in which they are to be executed. ROP gadgets, when chained together in this way, are Turing complete, and hence can be used to execute any kind of payload by using vulnerable program's very own code.

As a part of this project I propose a compiler pass that will instrument some code at every return instruction, that will be able to detect ROP attacks and other Buffer Overflow Attacks, and terminate the program if it is detected. This method prevents the ROP attack without much overhead to the compilation time, as well as the execution time, and being a purely software based solution, it does not require any specialized hardware for it's implementation.

2. COMPILER PASS TO PREVENT RETURN-ORIENTED PROGRAMMING (ROP) ATTACKS

2.1. *Working of the compiler pass*

The compiler pass will work on the Assembly code generate by GCC. It will then instrument some code, and finally use GCC to compile the resulting Assembly code into an executable. We will use the *–static* option to allow static linking, so that we can show that even if the attacker has access to several lines of code, it is still difficult for him to execute an ROP attack.

The compiler pass will look for every *ret* instructions in the Assembly Code. It will instrument code before this *ret* instruction, which will check if the return address is to a piece of code that called this function. More precisely, if in a function 'foo()' the return address is x, the instruction preceding the address x should be a call instruction. If it is not, the return address has been been overwritten. If we detect this security breach, we can terminate the program.

This pass can be implemented at compile time because we know the structure of the stack. Hence we know the location the of the return address pointer relative to the *esp* register, and we also know the opcode of the *call* instruction (we only look at the near calls at this point. The code can easily be extended to support far calls). There are a few versions of the *call* instruction. The first is with the opcode $e8$, and can be of length 3 or 5 bytes in total. The second is with opcode ff, followed by the values of 0,1,0 in the bits 3,4 and 5 of the next byte. The length of this ff instruction can be 2,4 or 6 bytes. Hence, if any of these above patterns exist in

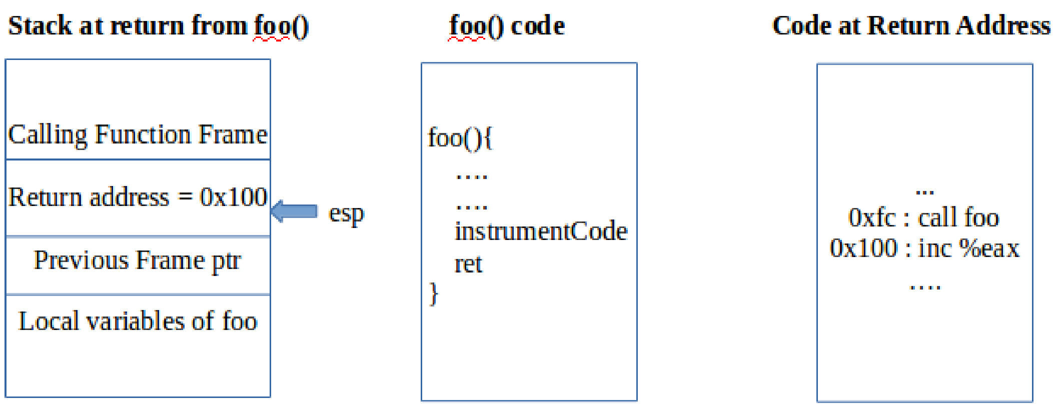

Figure 1. Example of the compiler pass operation

the bytes preceding the location of the return address, the return address is valid, otherwise it has been overwritten.

The figure below illustrates the main idea behind the pass. $foo()$ is the vulnerable function. The return address from $foo()$ on the stack is 0x100. So the job of the instrumented code is to verify that the few bytes before 0x100 form a valid *call* instruction. In this case it does. Hence we declare the return address as valid.

2.2. *If the Attacker knows about the existence of the Pass*

In the case of general Buffer overflow attacks, this protection can be circumvented if the attacker knows about the existence of the pass. If the buffer overflow overwrites the return address to point to the payload at address x, the attacker can modify the payload to include a call instruction just before this address x, and the attack would succeed because the instrumented code would treat it as a valid return address.

In the case of ROP attacks, you do not have complete control over the payload, since you are constructing gadgets from the instructions of the program. The protection now imposes a new condition that the return address location must have a *call* instruction before it. This drastically reduces the number of valid gadgets because there will be very few gadgets whose previous instruction is a call instruction. The reduced set of gadgets will not provide enough instructions to construct the required payload. Similarly, the 'return-to-libc' attacks will also be restricted to return to an address whose previous instruction is a *call* instruction.

This protection also holds good for ROP gadgets that are not aligned with instruction boundaries because the instrumented code will only interpret the bits before the return address to verify if it is a call instruction. It will not worry about the instruction boundaries.

2.3. *Design Decisions*

2.3.1. Running the Pass on Assembly code, and not on the source code or Intermediate Representation

It is possible that other compiler passes add some code to the source or IR. If our pass gets executed before these type of passes, it can happen that the stack structure gets modified or code is entered between the instrumentation and return statements, and our instrumentation is no more valid because the *esp* register points elsewhere. Hence the pass was chosen to run on assembly code. Also, LLVM IR cannot access architectural registers like the *esp* register. Additionally, programs written in any language that compiles to assembly at compile time can use this pass.

2.3.2. Not Checking if the call instruction is to the called function

It seems that the compiler pass can perhaps be improved if we do the further check that the bits before the return address are not only a *call* instruction, but itś operand is the current function address. This cannot be done because call instruction need not have a direct operand. That is, the function address to call may not be directly specified at compile time, but it may be decided dynamically at runtime. This especially happens in the case of function pointers or dynamically linked libraries. Hence we cannot instrument code in which the call instruction operand is indirect and known only at runtime.

3. IMPLEMENTATION AND RESULTS

There are 2 parts to the implementation. The first is the implementation of the actual pass. The second is counting the total number of gadgets available for an ROP attack before and after the pass gets implemented.

3.1. *Implementation of the actual Pass*

The concept of the pass has already been explained in section 2. The code is to be instrumented just before the *ret* instruction. Given below is the pseudo code for the same.

```
nextLocationOnStack = contents of esp ;
returnAddress = contents of nextLocationOnStack ;
if the contents 3 or 5 bytes before returnAddress is the opcode e8 then
    go to the ret statement ;
end
if the contents 2,4 or 6 bytes before returnAddress is the opcode ff, and the byte
  after ff has the bits number 3,4 and 5 of the next byte equal to 0,1 and 0
  respectively then
    go to the ret statement ;
end
/* If we reached this far, there is no valid call instruction before
   the return address location                                        */
Print error message and terminate program ;
```

Algorithm 1: Code instrumented by Compiler Pass

.

Dereferencing the *esp* register gives us the return address value. Dereferencing a few bytes before this value, allows us to examine the bytes before the return location in order to examine if it is a valid call instruction. The above code translates to roughly 80 assembly level instructions.

3.2. *Counting the Number of Valid and Invalid ROP Gadgets*

The total number of gadgets were calculated using the open source program https://github.com/JonathanSalwan/ROPgadget. It calculates all the possible gadgets at any given depth (depth is the size of the gadget). A modification was made to the source code to make the same checks as that specified in the implementation details in section 3.1. All gadgets which satisfy the check (of having a call instruction before them) are considered valid and those that do not are considered invalid. Each gadget is annotated with the tag of valid or invalid, and a total count of each type is computed.

By using the *–static* option while compiling with GCC, we can statically link LibC. On running the modified ROPgadget program (with a default depth of 10) on a small program with this static linking, we get 13730 unique gadgets, out of which only 216 are valid. Hence 98.42 percent of gadgets are invalid and only the valid gadgets can be used to construct the ROP attack after the Compiler pass is implemented. It is quite unlikely that a useful payload can be constructed with such few gadgets. If any of the invalid gadgets are used, the instrumented code will detect it and terminate the program.

4. RELATED WORK

ROPdefender (2) instruments the binary to have a shadow return address stack. All return addresses are duplicated here, and while returning, the return address is compared with this shadow return address.

(3) uses a technique called program shepherding to ensure that library code is entered only from the specified entry points

(4) uses the fact that ROP uses short instruction sequence ending in ret, and executes the gadgets contiguously in specific memory space, such as standard GNU libc. It uses a tool called DROP to dynamically identify such a usage pattern.

G-Free (1) encrypts the return address when placing it on the stack, and decrypts it while popping it from the stack. Hence, if the return address is overwritten, it will be decrypted to garbage on popping it. It also uses code rewriting techniques to eliminate all unintended free-branch opcodes.

References

(1)G-Free: defeating return-oriented programming through gadget-less binaries : ACSAC '10 Proceedings of the 26th Annual Computer Security Applications Conference, 2010

(2)M. W. Lucas Davi, Ahmad-Reza Sadeghi. Ropdefender: A detection tool to defend against return-oriented programming attacks. Technical report, Technical Report HGI-TR-2010-001.

(3)V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In Proceedings of the 11th USENIX Security Symposium, pages 191206, Berkeley, CA, USA, 2002. USENIX Association.

(4)P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In Lecture Notes in Computer Science, 2009