Return Oriented Programming

Coleman Kane
Coleman.Kane@ge.com

February 3, 2016

Introduction

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Return Oriented Programming is a mechanism to take advantage of memory corruption vulnerabilities that enable a user of a program to take control over the stack using specially-crafted program input. By experimenting with a program, you may craft special input that can cause addresses in the stack to be overwritten with data provided by your input.

Program Stack

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Below is a diagram of the "call stack". You'll notice that local variables, as well as restore addresses for ESP/RSP and EBP/RBP registers are stored in this structure.

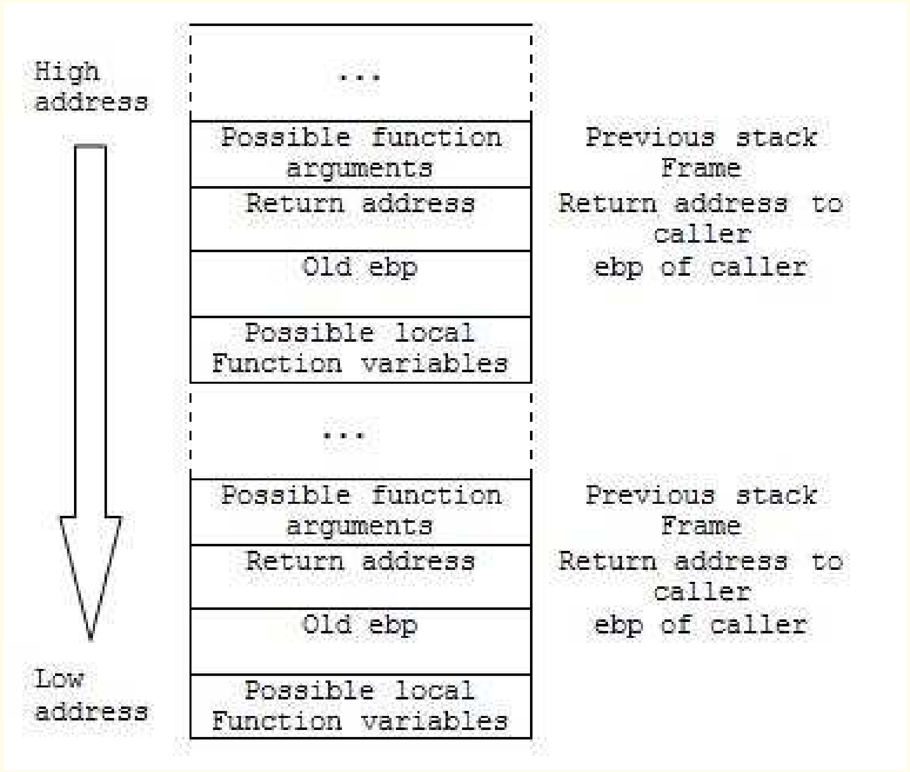

Figure 1: Stack Diagram

Example Vulnerability

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

```c
void
vuln_f(char *str) {
  char buffer[100];
  strcpy(buffer, str);
}
int main(int argc, char **argv) {
  vuln_f(argv[1]);
}
```

Vulnerability Description

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

At line 3, the author allocates (at least) 100 bytes on the stack to provide space for a local buffer to store transient data that's discarded after function return. The author uses `strcpy` to copy from the user-provided string into the buffer. The `strcpy` will copy data until it reaches a NULL (0x00) byte, regardless of buffer's size in the code. Since the function return address, old stack, and old ebp are all on the stack above the buffer, it is possible to provide a cmdline argument that contains arbitrary binary code to overwrite the return address and cause the program to return to a location chosen by the user, rather than the author.

Usage

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Using this exploit can help circumvent Data Execution Prevention (**DEP**). This is because you can control program execution flow without necessarily injecting your own new executable code at run-time. Thus, in order for ROP to work, all of the code that you wish to execute using the exploit must already exist in memory somewhere at the time of execution. For many Linux and Windows programs that load common functionality into memory using dynamic library linking at run-time, there are numerous possible options.

Fragments

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Since you can use the exploit to enter into any existing function, using the load address, you can also cause the program to enter into the middle of any existing code as well, thus executing a fragment. This may enable you to cause the portion of a system function to execute that you want, bypassing existing saftey mechanisms that may be built in to check data stored on the heap/stack.

Gadgets

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Building off of code fragments, another common approach to leveraging functions vulnerable to ROP is to employ what are known as **ROP Gadgets**. These are a nickname for extremely small snippets of assembly code that typically are 2-4 instructions in length, ending in a `RET` instruction. Rather than corrupting memory with a single return address to execute a function you wish, you can write multiple addresses into memory that point to a bunch of these gadgets, causing execution to cascade through a sequence of disparate code.

Gadget Examples

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

```
58 pop %eax
3C ret

33 C0 xor %eax,%eax
3C    ret

68 78 56 34 12 push 0x12345678
3C             ret
```

Attack Surface

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Specially-crafted command-line exploits require local access. A more common approach to leveraging these vulnerabilities is to deliver attacks targeting common end-user applications known to be vulnerable. Among those cited are frequently MSOffice, Java, Acrobat Reader, etc. Lesser applications known to be utilized by targets are also common (MS Vizio, media editing software, etc.).

Building Exploit for End-user Apps

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

In the case of end-user applications, you may be delivering a video, PDF file, or Office document that contains your exploit, but also contains a wealth of other information to make it appear "well formed" to the application. You may be able to introduce custom ROP gadgets into executable memory space by including fragments within other objects contained within the exploit payload. For instance, some frames from a video file may masquerade as corrupted but contain desired ROP gadgets.

Introduction
Program Stack
Example Vulnerability
Vulnerability Description
Usage
Fragments
Gadgets
Gadget Examples
Attack Surface
Building Exploit for End-user Apps
Using Applications

Using Applications

Researching the exploit target platform ahead of time in different states can help an exploit author identify where needed ROP gadgets will exist in memory at the time of exploit. This can be extremely helpful when, say, data provided through the exploit document cannot be coerced to load into executable memory space.