

Buffer Overflow Danger

Internet Break-In Statistics*

- 40% of compromised accounts/hosts are due to bad passwords.
- 35% are due to *buffer overflow exploits*.
- 10% are due to pancake mothballs.
- 8% are due to black helicopters.
- 2.600% are due to IP spoofing.

*** I made all these numbers up!**

Serious Note

- Try a web search for “buffer overflow exploit”.
- Check alt.2600, rootshell.com, antionline.com – you can find long lists of *exploits* based on buffer overflow.
- Even the original version of ssh had a problem! (after they made a big deal that there were no buffer overflow problems in their code).

The Problem

```
void foo(char *s) {  
    char buf[10];  
    strcpy(buf, s);  
    printf("buf is %s\n", s);  
}  
...  
foo("thisstringistolongforfoo");
```

Exploitation

- The general idea is to give programs (servers) very large strings that will overflow a buffer.
- For a server with sloppy code – it's easy to crash the server by overflowing a buffer (SEGV typically).
- It's sometimes possible to actually make the server do whatever you want (instead of crashing).

Background Necessary

- C functions and the stack.
- A little knowledge of assembly/machine language.
- How system calls are made (at the level of machine code level).
- **exec ()** system calls
- How to “guess” some key parameters.

CPU/OS dependency

- Building an exploit requires knowledge of the specific CPU and operating system of the target.
- I'll just talk about x86 and Linux, but the methods work for other CPUs and OSs.
- Some details are very different, but the concepts are the same.

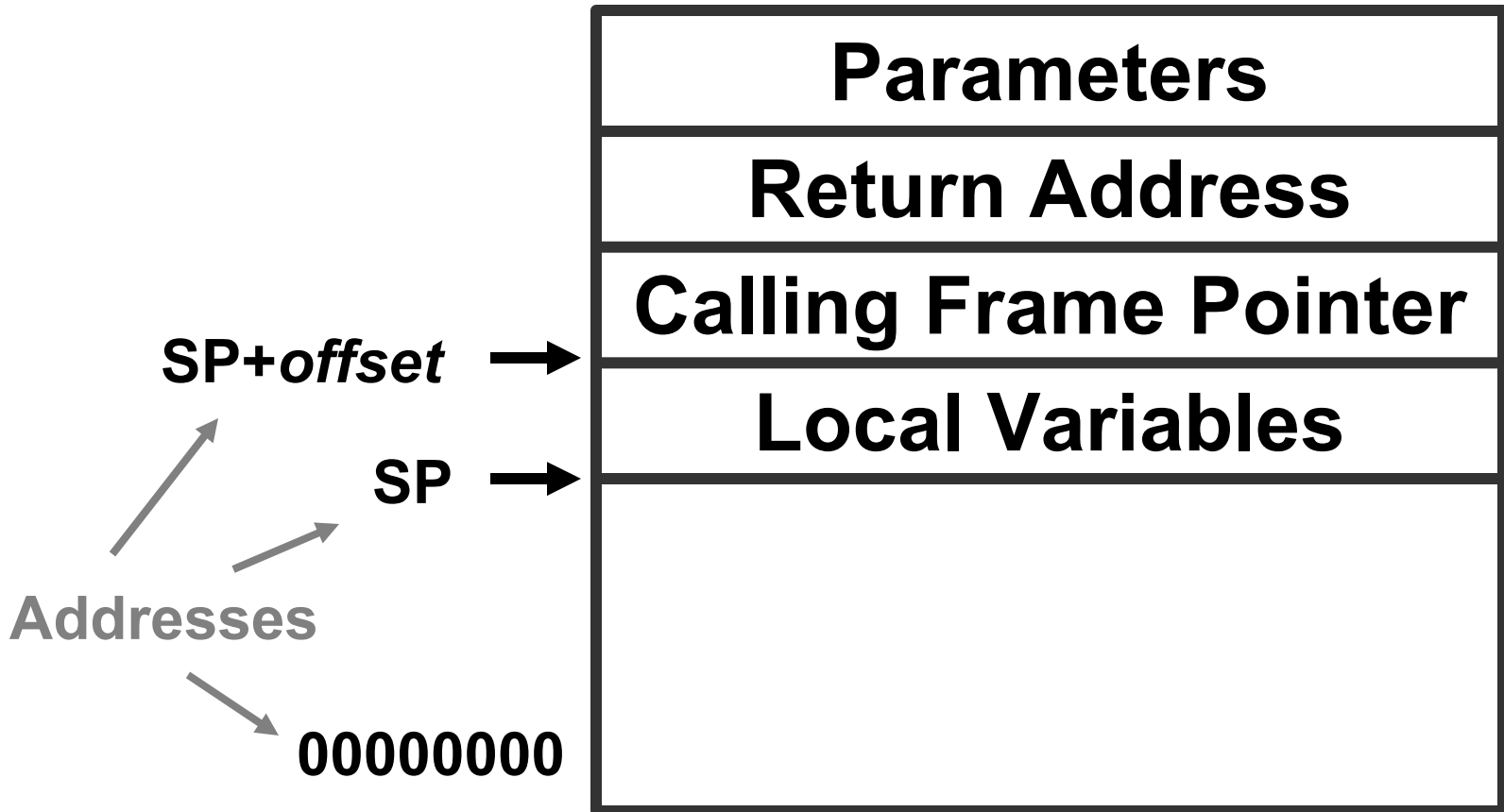
C Call Stack

- When a function call is made, the return address is put on the stack.
- Often the values of parameters are put on the stack.
- Usually the function saves the stack frame pointer (on the stack).
- Local variables are on the stack.

Stack Direction

- On Linux (x86) the stack grows from high addresses to low.
- Pushing something on the stack moves the Top Of Stack towards the address 0.

A Stack Frame



Sample Stack

18
<i>addressof(y=3) return address</i>
saved stack pointer
y
x
buf

```
x=2;  
foo(18);  
y=3;
```

```
void foo(int j) {  
    int x,y;  
    char buf[100];  
    x=j;  
    ...  
}
```

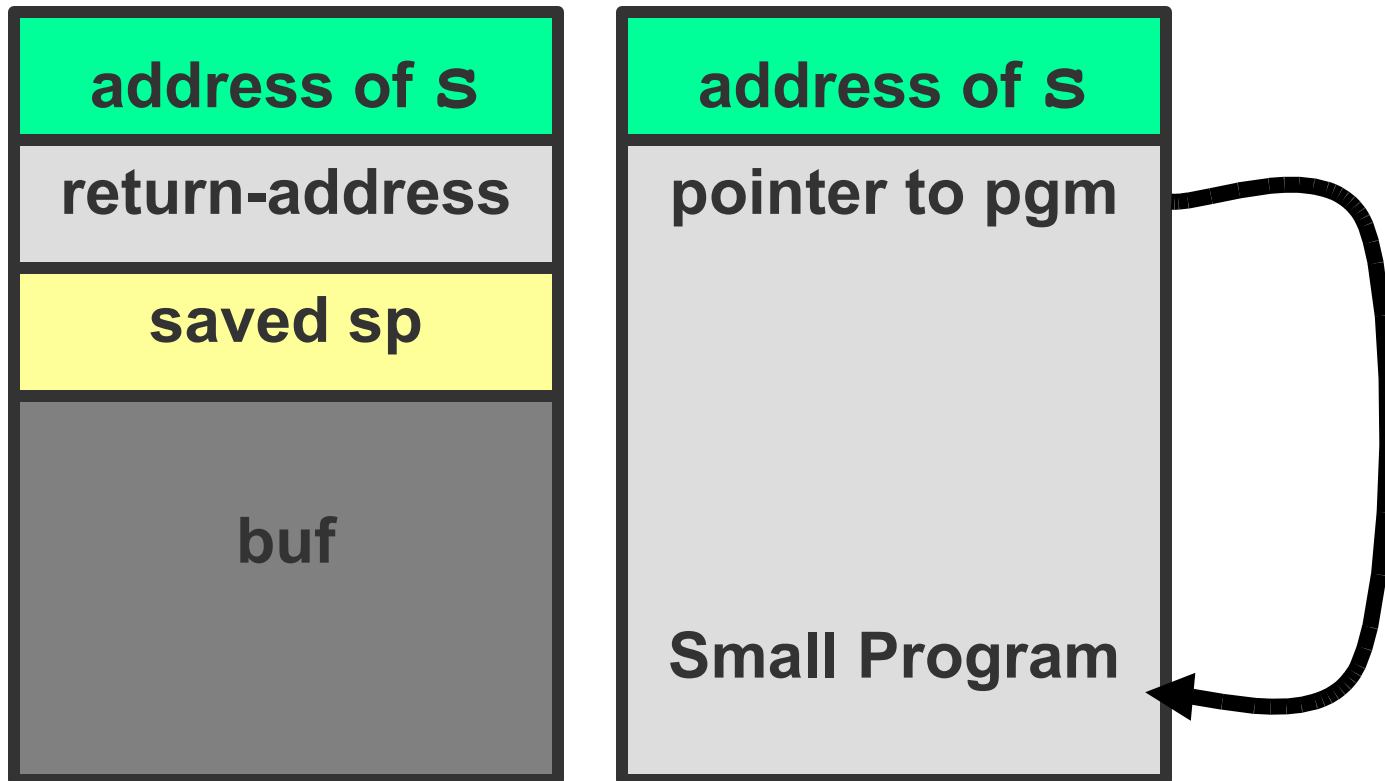
“Smashing the Stack”*

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

*taken from the title of an article in Phrack 49-7

Before and After

```
void foo(char *s) {  
    char buf[100];  
    strcpy(buf, s);  
    ...  
}
```



Issues

- How do we know what value the pointer should have (the new “return address”).
 - It’s the address of the buffer, but how do we know what address this is?
- How do we build the “small program” and put it in a string?

Guessing Addresses

- Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.
- An estimate is often good enough! (more on this in a bit).

Building the small program

- Typically, the small program stuffed in to the buffer does an `exec()` .
- Sometimes it changes the password db or other files...

`exec()`

- In Unix, the way to run a new program is with the `exec()` system call.
 - There is actually a *family* of `exec()` system calls...
 - This doesn't create a new process, it changes the current process to a new program.
 - To create a new process you need something else (`fork()`).

exec () example

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

```
char *args[] = {"/bin/ls", NULL};
```

```
void execls(void) {  
    execv("/bin/ls", args);  
    printf("I'm not printed\n");  
}
```

Generating a String

- You can take code like the previous slide, and generate machine language.
- Copy down the individual byte values and build a string.
- To do a simple exec requires less than 100 bytes.

A Sample Program/String

- Does an exec() of /bin/lS:

```
unsigned char cde[] =  
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"  
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"  
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"  
"\xcd\x80\x31\xdb\x89\xd8\x40xcd"  
"\x80\xe8\xdc\xff\xff\xff/bin/lS";
```

Some important issues

- The small program should be position-independent – able to run at any memory location.
- It can't be too large, or we can't fit the program and the new return-address on the stack!

Sample Overflow Program

```
unsigned char cde[] = "\xeb\x1f\...

void tst(void) {
    int *ret;
    ret = (int *)&ret+2; // pointer arith!
    (*ret) = (int) cde; //change ret addr
}

int main(void) {
    printf("Running tst\n");
    tst();
    printf("foo returned\n");
}
```

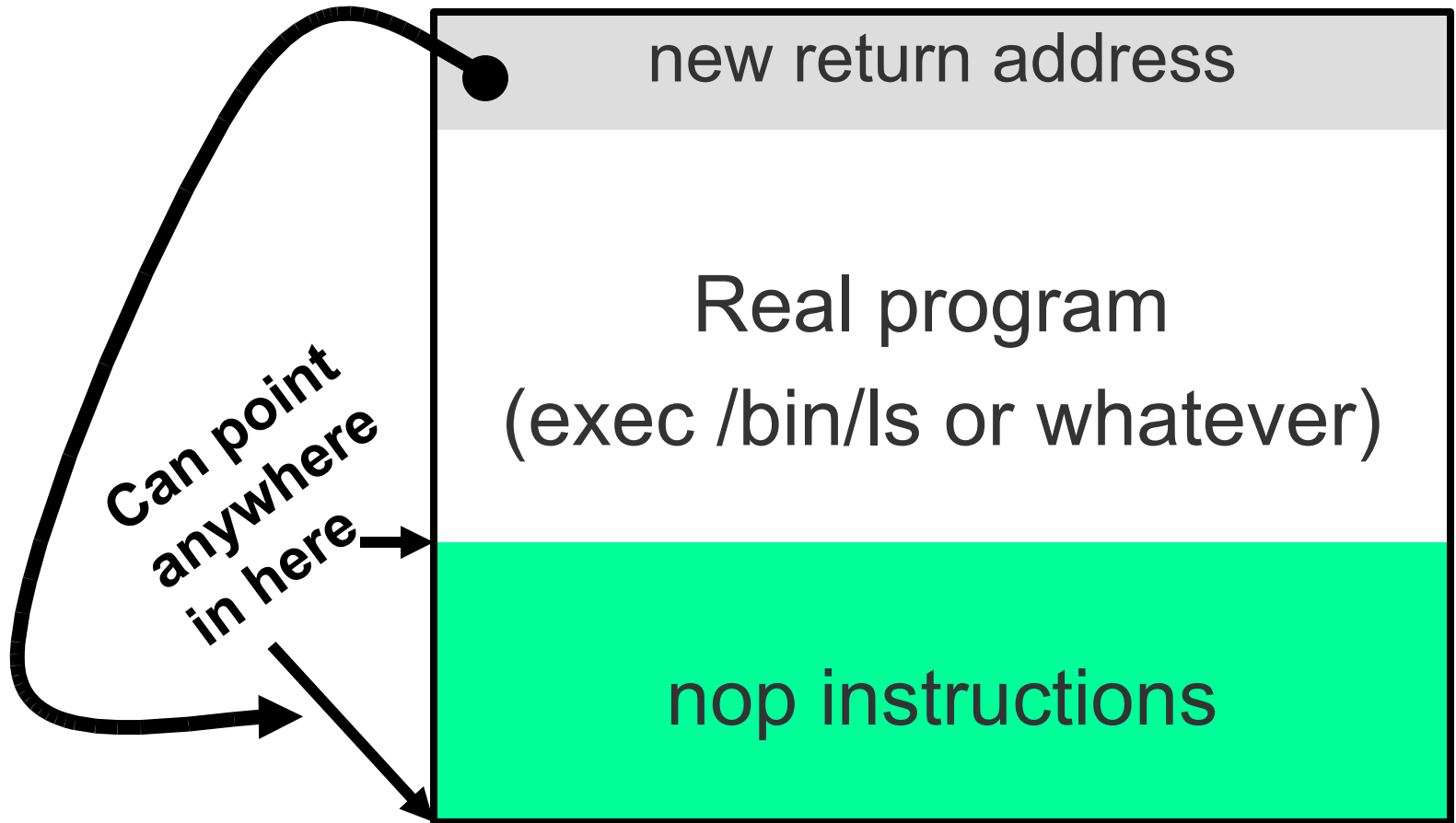
Attacking a real program

- Recall that the idea is to feed a server a string that is too big for a buffer.
- This string overflows the buffer and overwrites the return address on the stack.
- Assuming we put our small program in the string, we need to know its address.

NOPs

- Most CPUs have a *No-Operation* instruction – it does nothing but advance the instruction pointer.
- Usually we can put a bunch of these ahead of our program (in the string).
- As long as the new return-address points to a NOP we are OK.

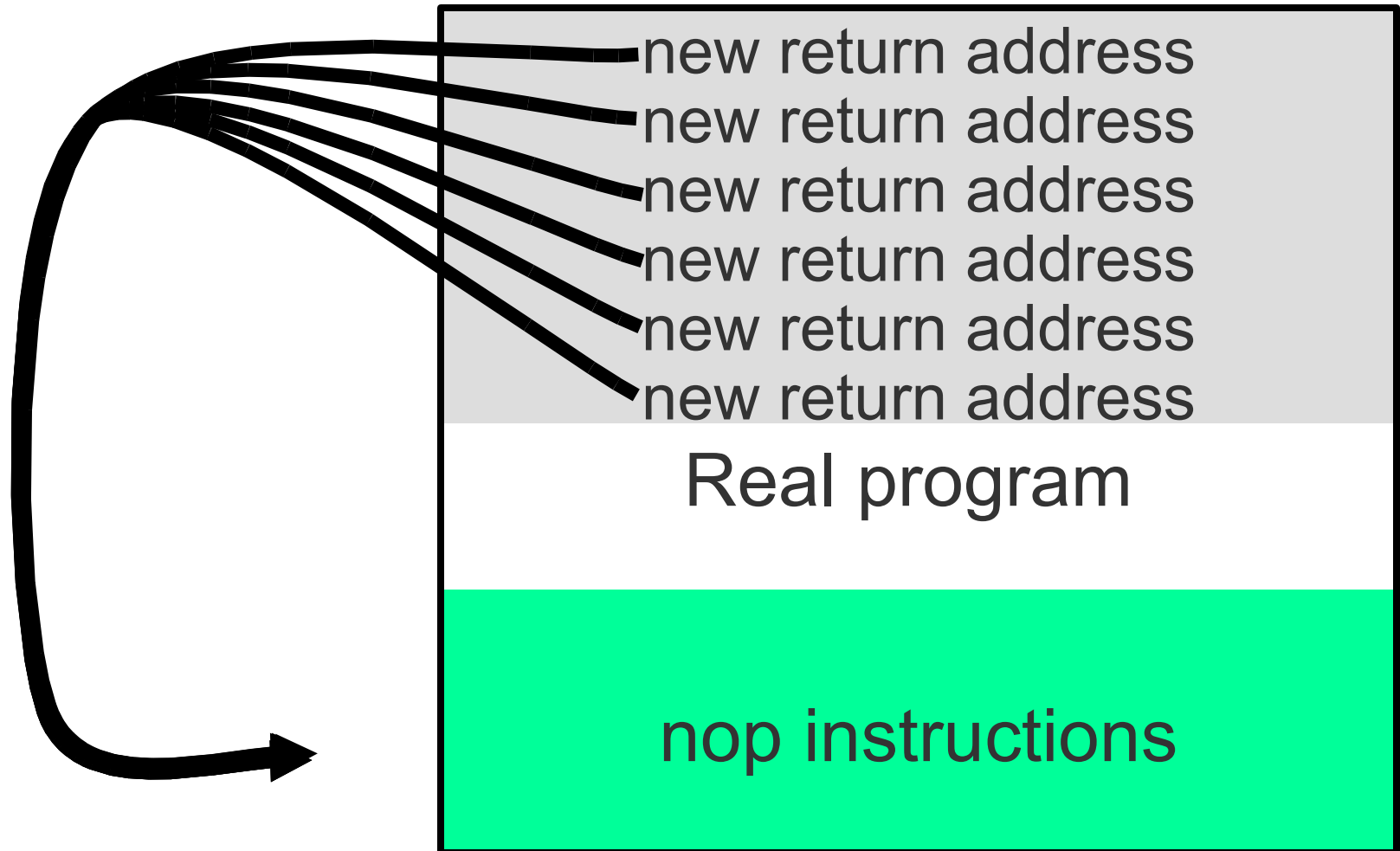
Using NOPs



Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.
- Put in a bunch of new return addresses!

Estimating the Location



vulnerable.c

```
void foo( char *s ) {
    char name[200];
    strcpy(name,s);
    printf("Name is %s\n",name);
}

int main(void) {
    char buf[2000];
    read(0,buf,2000);
    foo(buf);
}
```

genpgm.c

- genpgm.c was constructed to exploit the buffer overflow in vulnerable.c
- It allows the user to add an offset to a fixed “guess” of the address of the return-address on the stack.
- It writes (to stdout) a string that contains a bunch of return-addresses and a program that does: **exec /bin/ls.**

Testing

- `./genpgm 16 | ./vulnerable`
- Get ambitious! Change the program output by genpgm to `exec /bin/sh!`
- `(./genpgm; cat) | ./vulnerable`