

**Buffer Overflow Danger**

- Buffer Overflow 1

---

---

---

---

---

---

---

---

**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!**

CompOrg - Buffer Overflow 2

---

---

---

---

---

---

---

---

**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).

CompOrg - Buffer Overflow 3

---

---

---

---

---

---

---

---

## The Problem

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

CompOrg - Buffer Overflow

4

---

---

---

---

---

---

---

---

## 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).

CompOrg - Buffer Overflow

5

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

6

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

7

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

8

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

9

---

---

---

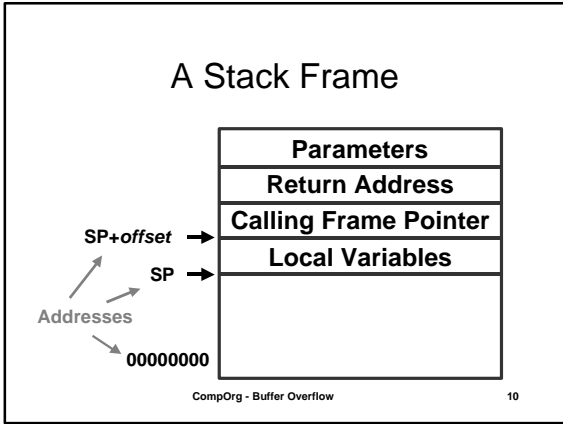
---

---

---

---

---




---

---

---

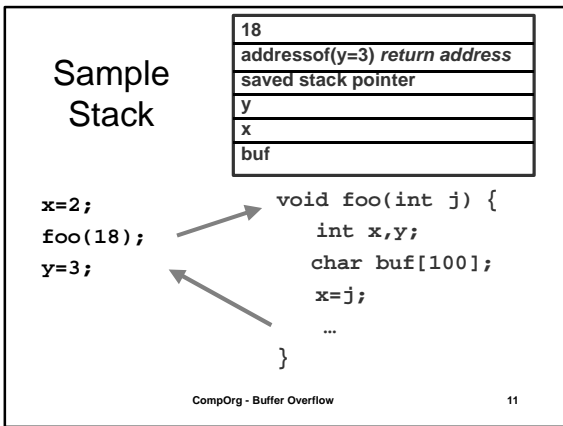
---

---

---

---

---




---

---

---

---

---

---

---

---

### “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

CompOrg - Buffer Overflow 12

---

---

---

---

---

---

---

---

Before and After

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

CompOrg - Buffer Overflow 13

---

---

---

---

---

---

---

---

### 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?

CompOrg - Buffer Overflow 14

---

---

---

---

---

---

---

---

### 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).

CompOrg - Buffer Overflow 15

---

---

---

---

---

---

---

---

## 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...

CompOrg - Buffer Overflow

16

---

---

---

---

---

---

---

---

## `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()`).

CompOrg - Buffer Overflow

17

---

---

---

---

---

---

---

---

## `exec()` example

```
#include <stdio.h>

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

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

CompOrg - Buffer Overflow

18

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

19

---

---

---

---

---

---

---

---

## 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";
```

CompOrg - Buffer Overflow

20

---

---

---

---

---

---

---

---

## 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!

CompOrg - Buffer Overflow

21

---

---

---

---

---

---

---

---

## Sample Overflow Program

```
unsigned char cde[] = "\xeb\x1f\...\n\nvoid tst(void) {\n    int *ret;\n    ret = (int *)&ret+2; // pointer arith!\n    (*ret) = (int) cde; //change ret addr\n}\n\nint main(void) {\n    printf("Running tst\\n");\n    tst();\n    printf("foo returned\\n");\n}
```

CompOrg - Buffer Overflow

22

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

23

---

---

---

---

---

---

---

---

## 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.

CompOrg - Buffer Overflow

24

---

---

---

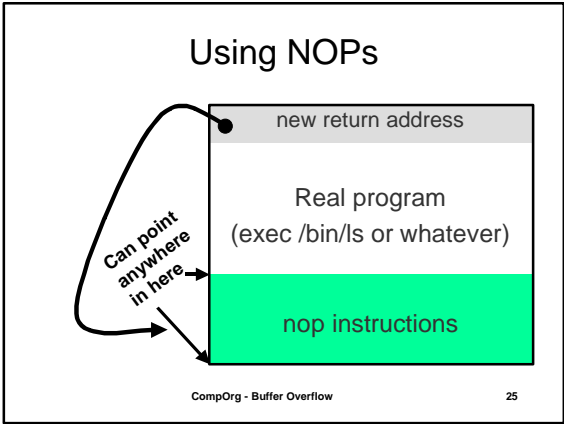
---

---

---

---

---




---

---

---

---

---

---

---

---

- ### 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!
- CompOrg - Buffer Overflow 26

---

---

---

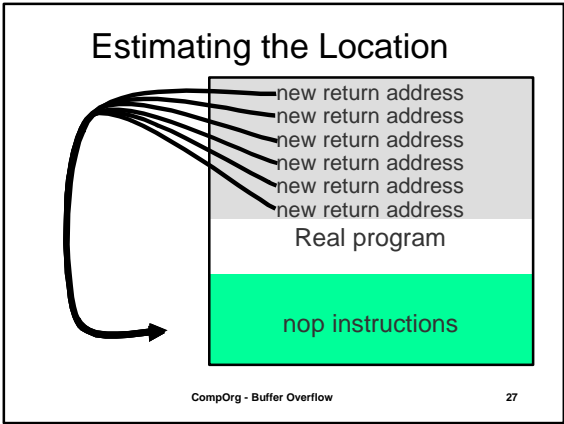
---

---

---

---

---




---

---

---

---

---

---

---

---

## 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);
}
```

CompOrg - Buffer Overflow

28

---

---

---

---

---

---

---

---

## genp gm . c

- genp gm . 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`.

CompOrg - Buffer Overflow

29

---

---

---

---

---

---

---

---

## Testing

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

CompOrg - Buffer Overflow

30

---

---

---

---

---

---

---

---