Access Control and Privacy Policies (3)

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

first lecture

first lecture

today

Smash the Stack for Fun ...

- "smashing the stack attacks" or
  "buffer overflow attacks"

- 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 by Elias Levy
  (also known as Aleph One):

  "Smashing The Stack For Fun and Profit"

  Issue 49, Article 14

The Problem

- The basic problem is that library routines in C look as follows:

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

- the resulting problems are often remotely exploitable
- can be used to circumvents all access control (botnets for further attacks)

Variants

There are many variants:

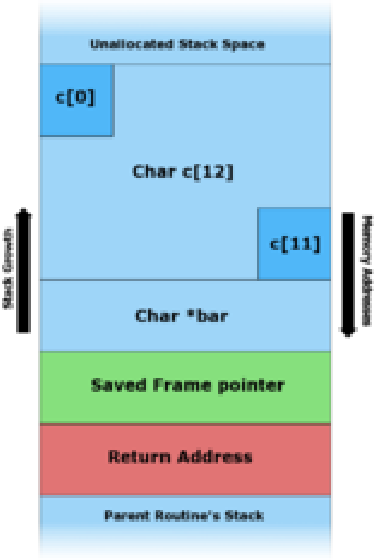
- return-to-lib-C attacks
- heap-smashing attacks
  (Slammer Worm in 2003 infected 90% of vulnerable systems within 10 minutes)
- “zero-days-attacks” (new unknown vulnerability)

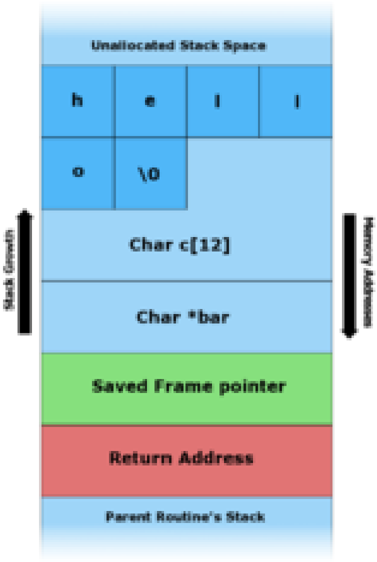
my_float is printed twice:

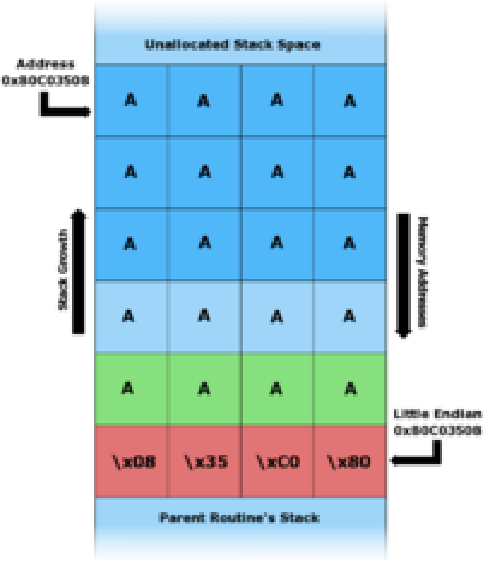
```
void foo (char *bar)
{
  float my_float = 10.5;    // in hex: \x41\x28\x00\x00
  char  buffer[28];

  printf(”my float value = %f\n”, my_float);
  strcpy(buffer, bar);
  printf(”my float value = %f\n”, my_float);
}

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


Unallocated Stack Space
c[0]
Char c[12]
c[11]
Char *bar
Saved Frame pointer
Return Address
Parent Routine's Stack
Stack Growth
Memory Addresses


Unallocated Stack Space
h
e
l
l
o
\0
Char c[12]
Char *bar
Saved Frame pointer
Return Address
Parent Routine's Stack
Stack Growth
Memory Addresses


Unallocated Stack Space
Address
0x80C03508
A A A A
A A A A
A A A A
A A A A
A A A A
\x08 \x35 \xC0 \x80
Stack Growth
Memory Addresses
Little Endian
0x80C03508
Parent Routine's Stack

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

void welcome() { printf(”Welcome to the Machine!\n”); exit(0); }
void goodbye() { printf(”Invalid identity, exiting!\n”); exit(1); }

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();
}
```

A programmer might be careful, but still introduce vulnerabilities:

```
// Since gets() is insecure and produces lots of warnings,
// I use my own input function instead.
char ch;
int i;

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

Payloads

- the idea is you store some code as part to the buffer
- you then override the return address to execute this payload
- normally you start a root-shell

Payloads

- the idea is you store some code as part to the buffer
- 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"

Payloads (2)

- 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;
  }
}
```

Format String Vulnerability

string is nowhere used:

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

// a program that just prints the argument
// on the command line
//
// try and run it with %s


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 BO Atta

- use safe library functions
- 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)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)
- Monitoring (detect attacks)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)
- Monitoring (detect attacks)
- Privacy, confidentiality, anonymity (to protect secrets)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)
- Monitoring (detect attacks)
- Privacy, confidentiality, anonymity (to protect secrets)
- Authenticity (needed for access control)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)
- Monitoring (detect attacks)
- Privacy, confidentiality, anonymity (to protect secrets)
- Authenticity (needed for access control)
- Integrity (prevent unwanted modification or tampering)

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks (traceability and auditing of security-relevant actions)
- Monitoring (detect attacks)
- Privacy, confidentiality, anonymity (to protect secrets)
- Authenticity (needed for access control)
- Integrity (prevent unwanted modification or tampering)
- Availability and reliability (reduce the risk of DoS attacks)

Homework

- Assume format string attacks allow you to read out the stack. What can you do with this information?

- Assume you can crash a program remotely. Why is this a problem?