Access Control and Privacy Policies (3)

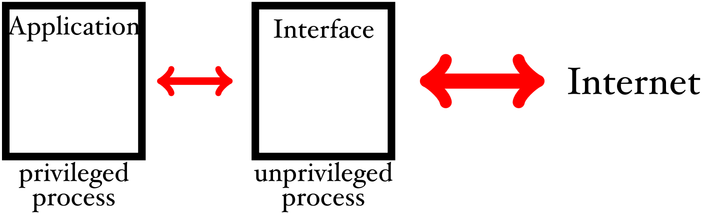
Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)
Slides:	KEATS (also home work is there)

first lecture

first lecture

today

Network Applications: Privilege Separation



- the idea is make the attack surface smaller and mitigate the consequences of an attack

Access Control in Unix

- access control provided by the OS
- authenticate principals (login)
- mediate access to files, ports, processes according to roles (user ids)
- roles get attached with privileges

> The principle of least privilege:
> programs should only have as much privilege as they need

Process Ownership

- access control in Unix is very coarse

$$\frac{\text{root}}{\text{user}_1\ \text{user}_2\ \ldots\text{www, mail, lp}}$$

root has UID = 0

Process Ownership

- access control in Unix is very coarse

$$\frac{\text{root}}{\text{user}_1 \ \text{user}_2 \ \ldots \text{www, mail, lp}}$$

root has UID = 0
you also have groups that can share access to a file
but it is difficult to exclude access selectively

Access Control in Unix (2)

- privileges are specified by file access permissions ("everything is a file")
- there are 9 (plus 2) bits that specify the permissions of a file

```
$ ls - la
-rwxrw-r--    foo_file.txt
```

Login Process

- login processes run under UID = 0

$$\text{ps -axl} \mid \text{grep login}$$

- after login, shells run under UID = user (e.g. 501)

$$\text{id cu}$$

Login Process

- login processes run under UID = 0

  ps -axl | grep login

- after login, shells run under UID = user (e.g. 501)

  id cu

- non-root users are not allowed to change the UID — would break access control
- but needed for example for passwd

Setuid and Setgid

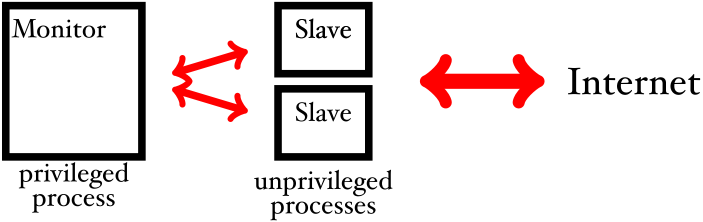
The solution is that unix file permissions are 9 + 2 Bits: Setuid and Setgid Bits

- When a file with setuid is executed, the resulting process will assume the UID given to the owner of the file.
- This enables users to create processes as root (or another user).

- Essential for changing passwords, for example.

chmod 4755 fobar_file

Privilege Separation in OpenSSH



- pre-authorisation slave
- post-authorisation

- 25% codebase is privileged, 75% is unprivileged

Network Applications

ideally network application in Unix should be designed as follows:

- need two distinct processes
  - one that listens to the network; has no privilege
  - one that is privileged and listens to the latter only (but does not trust it)
- to implement this you need a parent process, which forks a child process
- this child process drops privileges and listens to hostile data
- after authentication the parent forks again and the new child becomes the user

Famous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...

Famous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...

Famous 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 ...

Famous 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)

Famous Security Flaws in Unix

- lpr unfortunately runs with root privileges; you had the option to delete files after printing ...
- for de… (FreeBSD) Unix …ides a “cor…
- mkdir … 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)

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

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

   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**

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

A Float 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;
}
```

The Problem

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

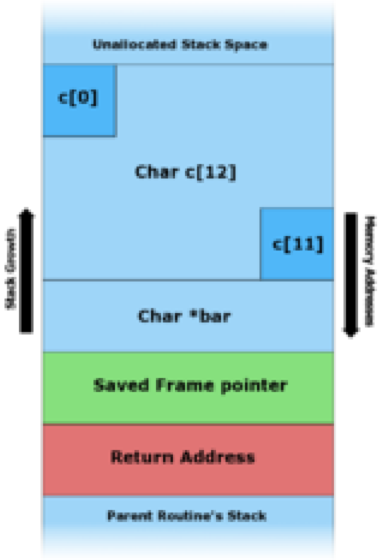
```
void strcpy(char *src, char *dst) {
  int i = 0;
  while (src[i] != ”\0”) {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

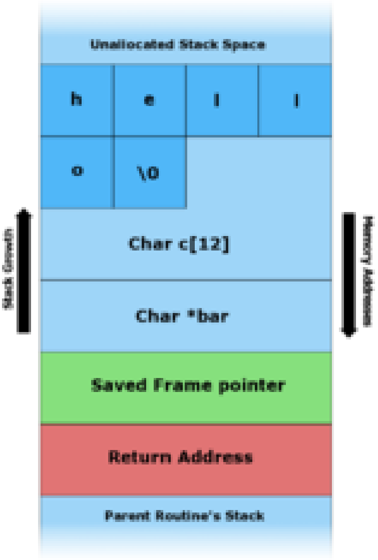
- the resulting problems are often remotely exploitable
- can be used to circumvents all access control (for grooming botnets for further attacks)

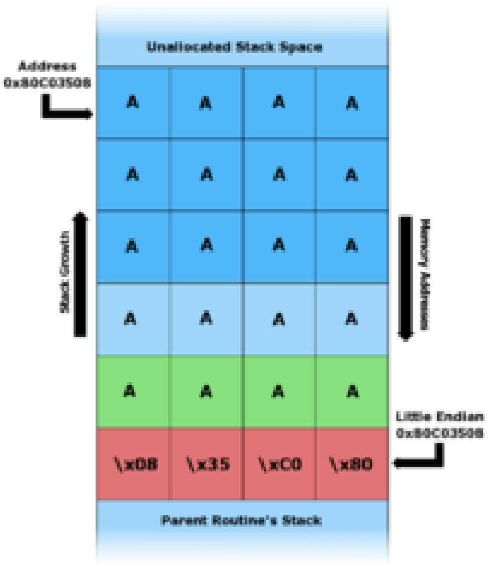
Variants

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)


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


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
Stack Growth
Memory Addresses
Little Endian
0x80C03508
\x08 \x35 \xC0 \x80
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:

```c
// 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 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 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:

$$\text{xorl \%eax, \%eax}$$

```
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:

```
#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 Attacks

- 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?