Vulnerability Analysis (IV): Program Verification

Slide credit: Vijay D'Silva

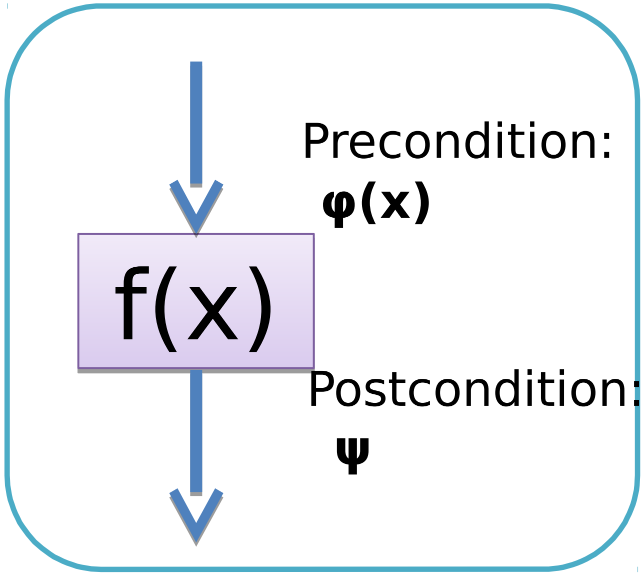
Program Verification

Program Verification

- How to prove a program free of buffer overflows?
  - Precondition
  - Postcondition
  - Loop invariants

Precondition

- Precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`
  - If any precondition is not met, f() may not behave correctly
  - Callee may freely assume obligation has been met
- The concept similarly holds for any statement or block of statements



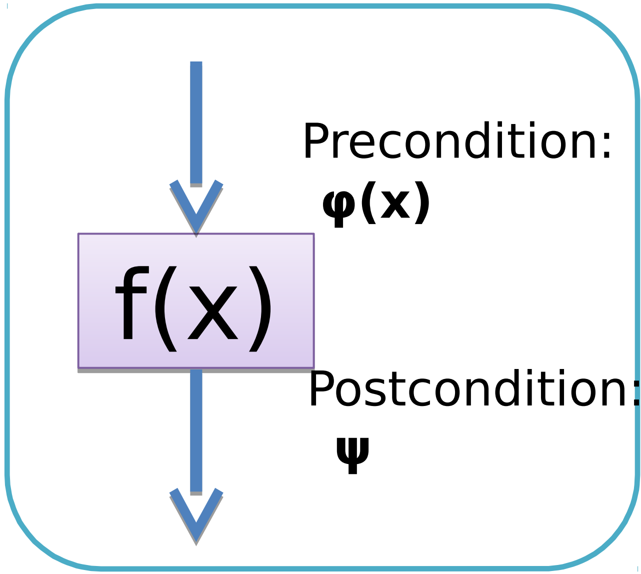
Precondition Example

- Precondition:
  - fp points to a valid location in memory
  - fp points to a file
  - the file that fp points to contains at least 4 characters
  - …

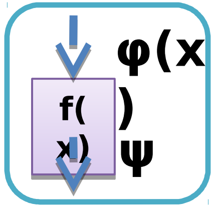
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘\n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Postcondition

- *Postcondition* for `f()`
  - An assertion that holds when `f()` returns
  - `f()` has obligation of ensuring condition is true when it returns
  - Caller may assume postcondition has been established by `f()`



Postcondition Example

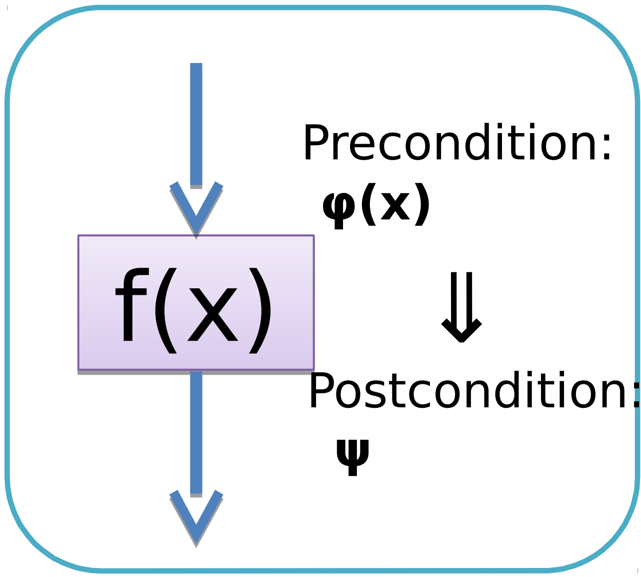


- Postcondition:
  - *buf* contains no uppercase letters
  - (return 0) ⇒(cmd[0..3] == “GET “)

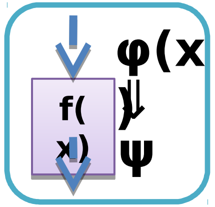
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```

Proving Precondition ⇒ Postcondition

- Given preconditions and postconditions
  - Specifying what obligations caller has and what caller is entitled to rely upon
- Verify: No matter how function is called,
  - if precondition is met at function's entrance,
  - then postcondition is guaranteed to hold upon function's return



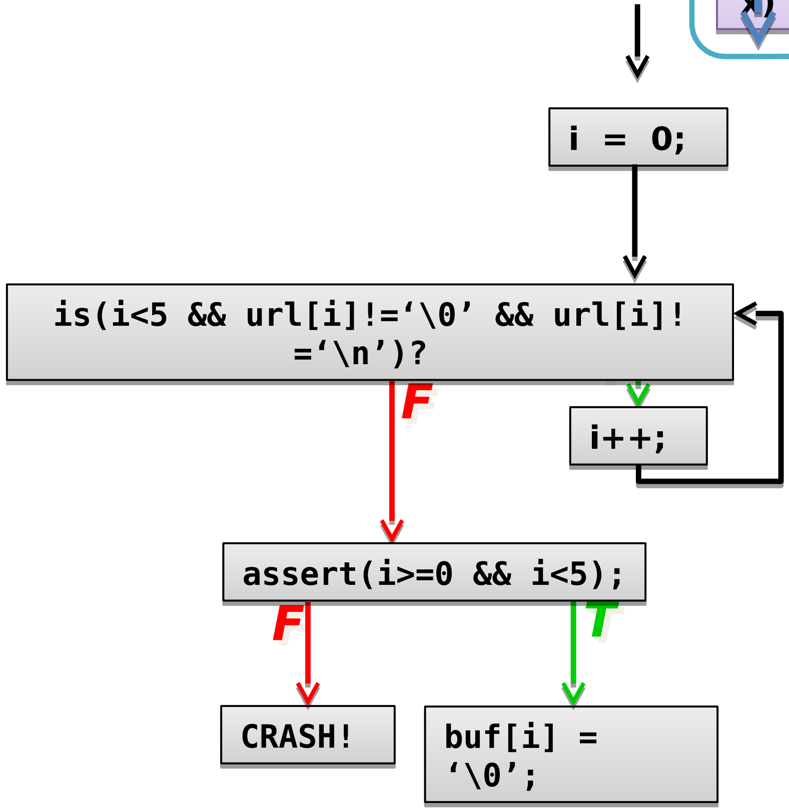
Proving Precondition ⇒ Postconditio



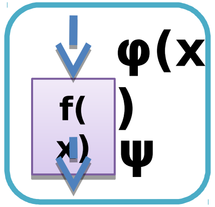
- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Use logical reasoning
- Requirement:
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one

We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.

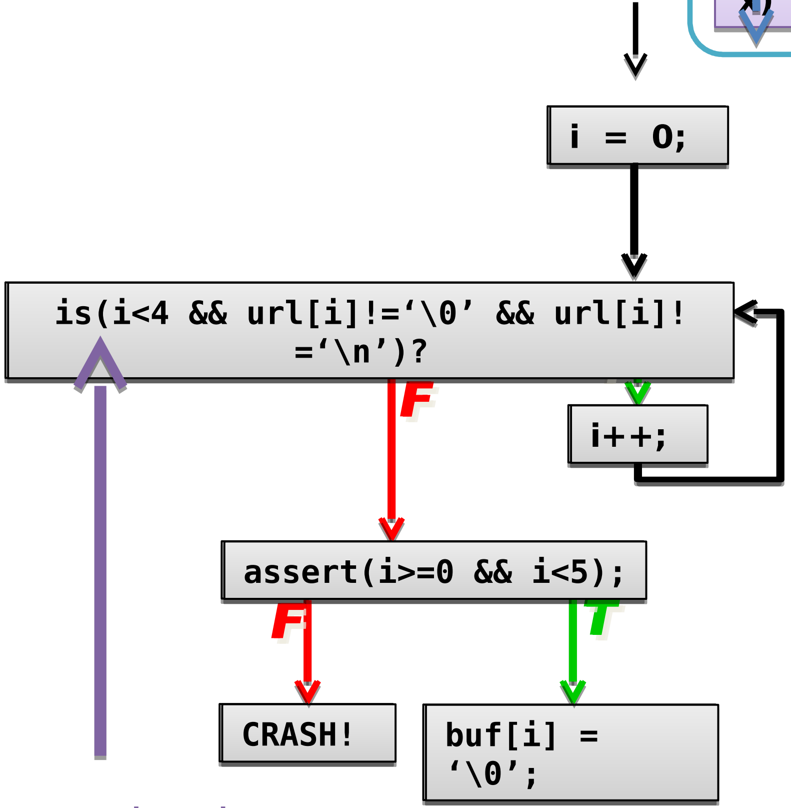
```
int parse(FILE *fp) {
  char cmd[256], *url, buf[5];
  fread(cmd, 1, 256, fp);
  int i, header_ok = 0;
  if (cmd[0] == ‘G’)
    if (cmd[1] == ‘E’)
      if (cmd[2] == ‘T’)
        if (cmd[3] == ‘ ’)
          header_ok = 1;
  if (!header_ok) return -1;
  url = cmd + 4;
  i=0;
  while (i<5 && url[i]!=‘\0’ && url[i]!=‘n’) {
    buf[i] = tolower(url[i]);
    i++;
  }
  assert(i>=0 && i <5);
  buf[i] = ‘\0’;
  printf(“Location is %s\n”, buf);
  return 0; }
```



We'll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.



```
1:int parse(FILE *fp) {
2:  char cmd[256], *url, buf[5];
3:  fread(cmd, 1, 256, fp);
4:  int i, header_ok = 0;
5:  if (cmd[0] == 'G')
6:    if (cmd[1] == 'E')
7:      if (cmd[2] == 'T')
8:        if (cmd[3] == ' ')
9:          header_ok = 1;
10:  if (!header_ok) return -1;
11:  url = cmd + 4;
12:  i=0;
13:  while (i<4 && url[i]!='\0' && url[i]!='n')
{
14:    buf[i] = tolower(url[i]);
15:    i++;
16:  }
17:  assert(i>=0 && i <5);
18:  buf[i] = '\0';
19:  printf("Location is %s\n", buf);
20:  return 0; }
```



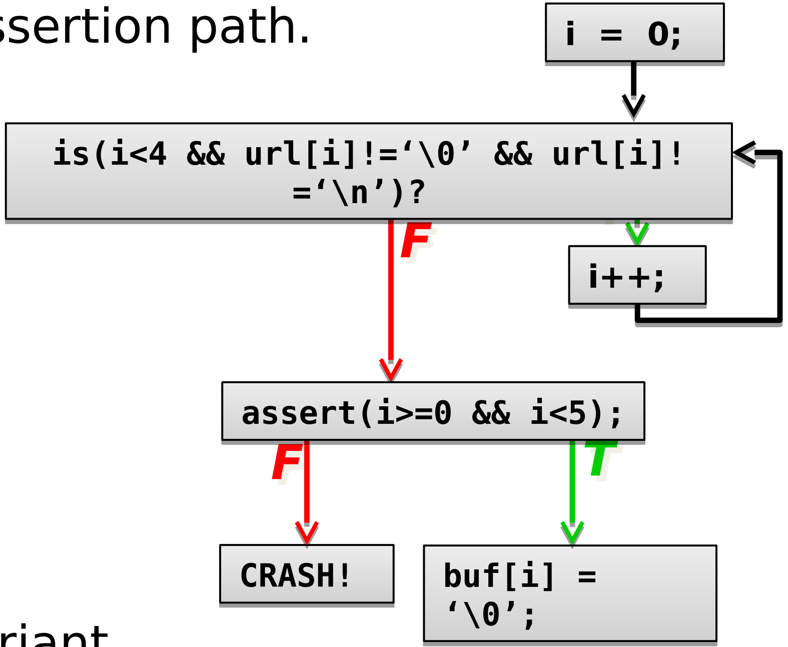
Bug Fixed!

Dawn Song

We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists…

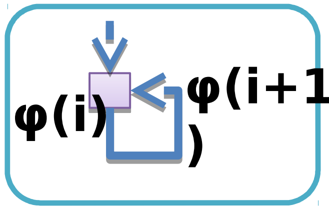
```
int parse(FILE *fp) {
 char cmd[256], *url, buf[5];
 fread(cmd, 1, 256, fp);
 int i, header_ok = 0;
 if (cmd[0] == ‘G’)
  if (cmd[1] == ‘E’)
   if (cmd[2] == ‘T’)
    if (cmd[3] == ‘ ’)
     header_ok = 1;
 if (!header_ok) return -1;
 url = cmd + 4;
 i=0;
 while (i<4 && url[i]!=‘\0’ && url[i]!=‘n’) {
  buf[i] = tolower(url[i]);
  i++;
 }
 buf[i] = ‘\0’;
 printf(“Location is %s\n”, buf);
 return 0; }
```

…So assuming fp points to a file that begins with “GET “, we want to show that *parse* never goes down the false assertion path.

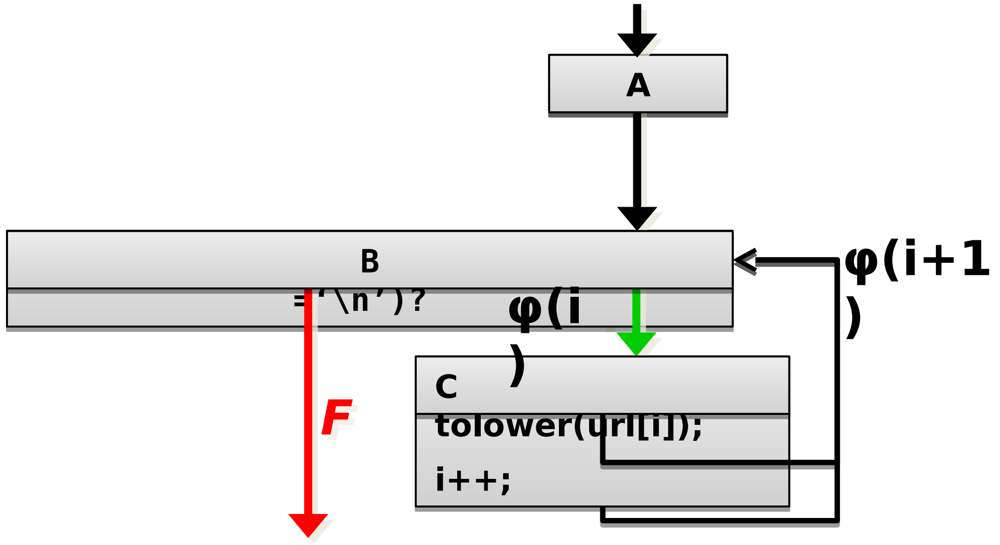


ut first, we will need the concept of loop invariant.

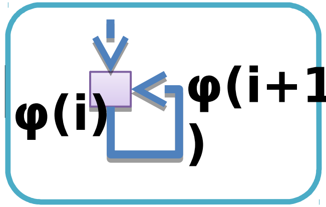
Loop Invariant and Induction



- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - Both a pre- and post-condition for the loop body



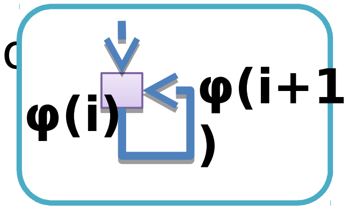
Loop Invariant and Inductio

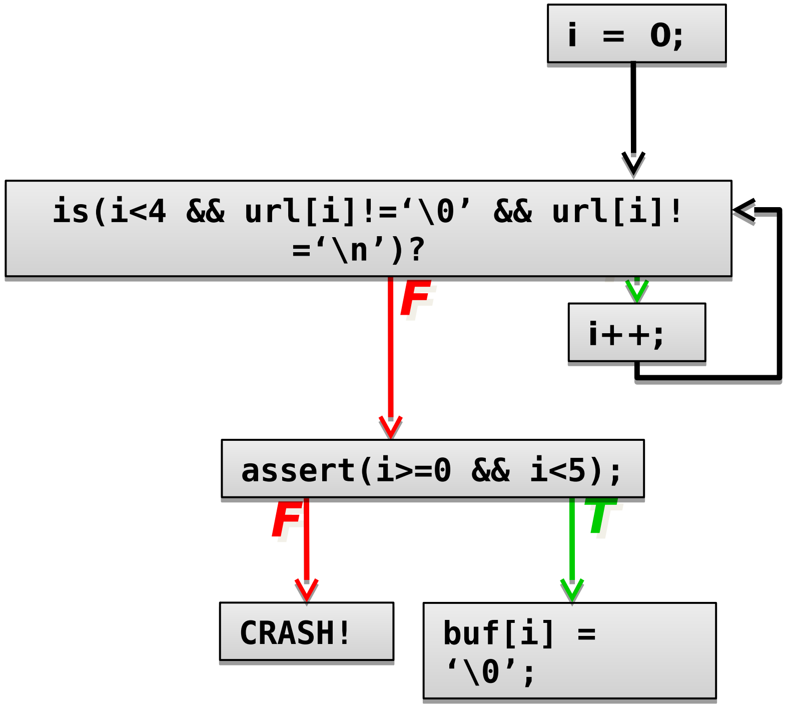


- To verify:
  - Base Case: Prove true for first iteration: **φ(0)**
  - Inductive step: Assume **φ(i)** at the beginning of the loop. Prove **φ(i+1)** at the start of the next iteration.

Try with our familiar example, proving that (0≤i<5) after the loo terminates:

LOOP INVARIANT: `/* φ(i) = (0≤i<5) */`





```
i = 0;
is(i<4 && url[i]!='\0' && url[i]!='\n')?
i++;
assert(i>=0 && i<5);
CRASH!
buf[i] = '\0';
```

Base Case:

```
/* φ(0) = (0≤0<5) */
```

Inductive Step:

```
/* assume(0≤i<5)at the beginning of the loop

/* for (0≤i<4), clearly (0≤i+1<5) */

  /* (i=5) is not a possible case since
     that would fail the looping predicate */

 /* ⇒ (0≤i+1<5) at the end of the loop */

  /* ⇒ parse never fails the assertion */
```

Function Post-/Pre-Conditions

- For every function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function `f()` by looking at only its code and the annotations on every function `f()` calls
    - Can ignore code of all other functions and functions called transitively
  - Makes reasoning about `f()` an almost purely local activity

Dafny

- A programming language with built-in specification constructs.
- A static program verifier to verify the functional correctness of programs.
- Powered by Boogie and Z3.
- Available here: http://rise4fun.com/dafny/

Documentation

- Pre-/post-conditions serve as useful documentation
  - To invoke Bob's code, Alice only has to look at pre- and post-conditions – she doesn't need to look at or understand his code
- Useful way to coordinate activity between multiple programmers:
  - Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  - Alice and Bob can negotiate the interface (and responsibilities) between their code at design time

Preventing Security Vulnerabilities

- Identify implicit requirements code must meet
  - Must not make out-of-bounds memory accesses, deference null pointers, etc.

- Prove that code meets these requirements
  - Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds

Preventing Security Vulnerabilities

- How easy it is to prove a certain property of code depends on how code is written
  - Structure your code to make it easy to prove

Security Architecture and Principles

Access Control & Capabilities

Access Control

- Some resources (files, web pages, …) are sensitive.
- How do we limit who can access them?
- This is called the access control problem

Access Control Fundamentals

- Subject = a user, process, …
  - (someone who is accessing resources)
- Object = a file, device, web page, …
  - (a resource that can be accessed)
- Policy = the restrictions we’ll enforce
- access(S, O) = true
  - if subject S is allowed to access object O

Access control matrix

[Lampson]

Objects

Subjects

	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

Two implementation concepts

- Access control list (ACL)
  - Store column of matrix with the resource
- Capability
  - User holds a "ticket" for each resource
  - Two variations
    - store row of matrix with user, under OS control
    - unforgeable ticket in user space

	File 1	File 2	…
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
…			
User m	Read	write	write

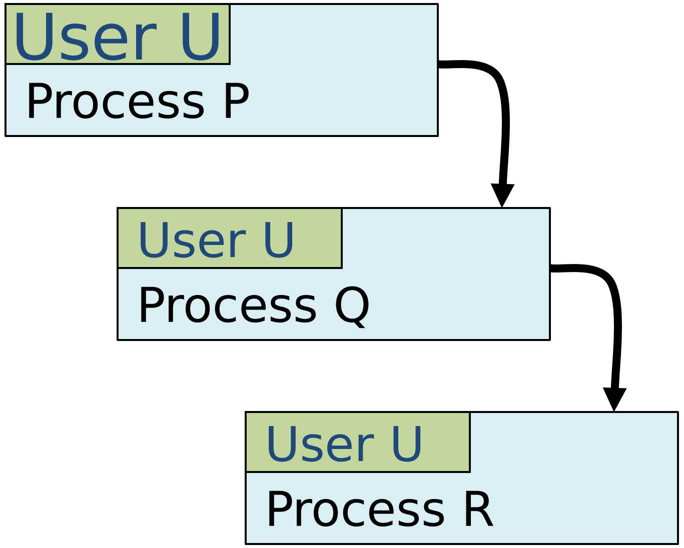
Access control lists are widely used, often with groups

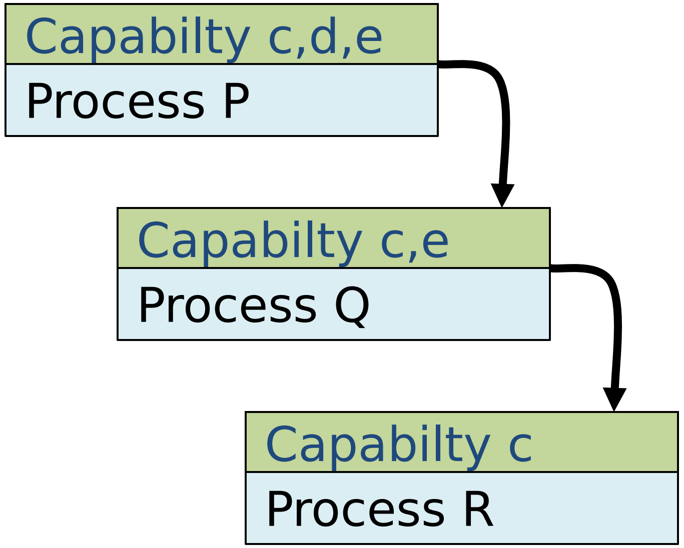
Some aspects of capability concept are used in many systems

ACL vs Capabilities

- Access control list
  - Associate list with each object
  - Check user/group against list
  - Relies on authentication: need to know user
- Capabilities
  - Capability is unforgeable ticket
    - Random bit sequence, or managed by OS
    - Can be passed from one process to another
  - Reference monitor checks ticket
    - Does not need to know identify of user/process

ACL vs Capabilities



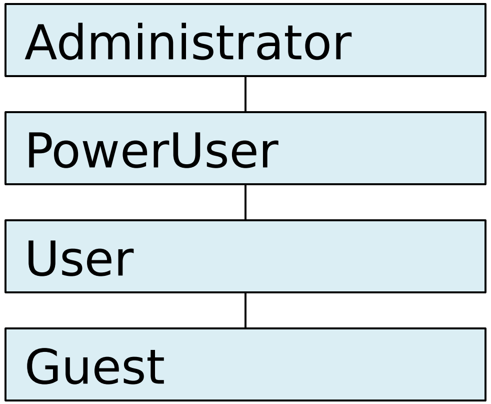


ACL vs Capabilities

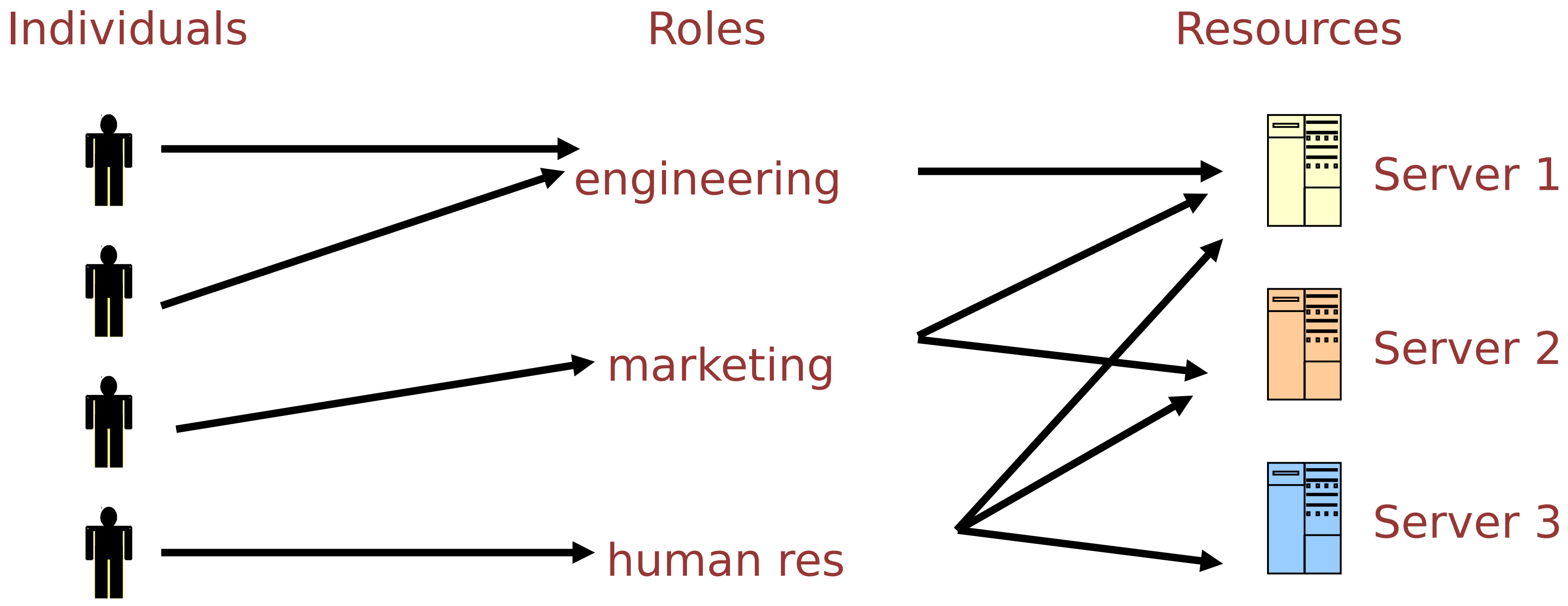
- Delegation
  - Cap: Process can pass capability at run time
  - ACL: Try to get owner to add permission to list?
    - More common: let other process act under current user
- Revocation
  - ACL: Remove user or group from list
  - Cap: Try to get capability back from process?
    - Possible in some systems if appropriate bookkeeping
      - OS knows which data is capability
      - If capability is used for multiple resources, have to revoke all or none …
    - Indirection: capability points to pointer to resource
      - If C $\rightarrow$ P $\rightarrow$ R, then revoke capability C by setting P=0

Roles (also called Groups)

- Role = set of users
  - Administrator, PowerUser, User, Guest
  - Assign permissions to roles; each user gets permission
- Role hierarchy
  - Partial order of roles
  - Each role gets permissions of roles below
  - List only new permissions given to each role



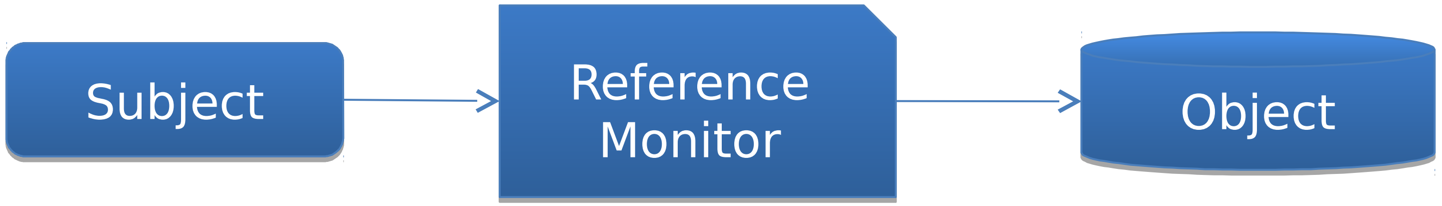
Role-Based Access Control



Advantage: user’s change more frequently than roles

Reference Monitor

- A reference monitor is responsible for mediating all access to data



- Subject cannot access data directly; operations must go through the reference monitor, which checks whether they are OK.

Criteria for a reference monitor

Ideally, a reference monitor should be:

- Unbypassable: all accesses go through the reference monitor (also called complete mediation)
- Tamper-resistant: attacker cannot subvert or take control of the reference monitor (e.g., no code injection)
- Verifiable: reference monitor should be simple enough that it’s unlikely to have bugs

Dawn Song

Non-Language-Specific Vulnerabilities

```
procedure withdrawal(w)
    // contact central server to get balance
    1. let b := balance

    2. if b < w, abort

    // contact server to set balance
    3. set balance := b - w

    4. dispense $w to user
```

Non-Language-Specific Vulnerabilities

```
// Part of a setuid program
if (access("file", W_OK) != 0) {
   exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

access("file", W_OK)	Returns 0 if the user invoking the program has write access to "file" (it checks the real uid, the actual id of the user, as opposed to the effective uid, the id associated with the process)
open("file", O_WRONLY)	Returns a handle to "file" to be used for writing only
write(fd, buffer ...)	Writes the contents of buffer to "file"

Time-of-Check-to-Time-of-Use (TOCTTOU)

```
// Part of a setuid program
if (access("file", W_OK) != 0) {
   exit(1);
}
```

```
// After the access check
symlink("/etc/passwd", "file");
// Before the open, "file"
   points to the password
   database
```

```
fd = open("file", O_WRONLY);
write(fd, buffer,
sizeof(buffer));
```

access("file", W_OK)	Returns 0 if the user invoking the program has write access to "file" (it checks the real uid, the actual id of the user, as opposed to the effective uid, the id associated with the process)
open("file", O_WRONLY)	Returns a handle to "file" to be used for writing only
write(fd, buffer …)	Writes the contents of buffer to "file"
symlink("/etc/passwd", "file")	Creates a symlink from "file" to "/etc/passwd". A symbolic link is a reference to another file, so in this case the attacker causes "file" (which they have privileges for) to point to "/etc/passwd". The program then opens "/etc/passwd" instead of "file".

The Flaw?

- Code assumes FS is unchanged between `access()` and `open()` calls – Never assume anything…
- An attacker could change file referred to by “`file`” in between `access()` and `open()`
  - Eg. `symlink(“/etc/passwd”, “file”)`
  - Bypasses the check in the code!
  - Although the user does not have write privileges for `/etc/passwd`, the program does (and the attacker has privileges for `file`, so they are allowed to create the symbolic link)
  - Time-Of-Check To Time-Of-Use (TOCTTOU) vulnerability
  - Meaning of `file` changed from time it is checked (`access()`) and time it is used (`open()`)

TOCTTOU Vulnerability

- In Unix, often occurs with file system calls because system calls are not atomic
- But, TOCTTOU vulnerabilities can arise anywhere there is mutable state shared between two or more entities
  - Example: multi-threaded Java servlets and applications are at risk for TOCTTOU

Minimize TCB

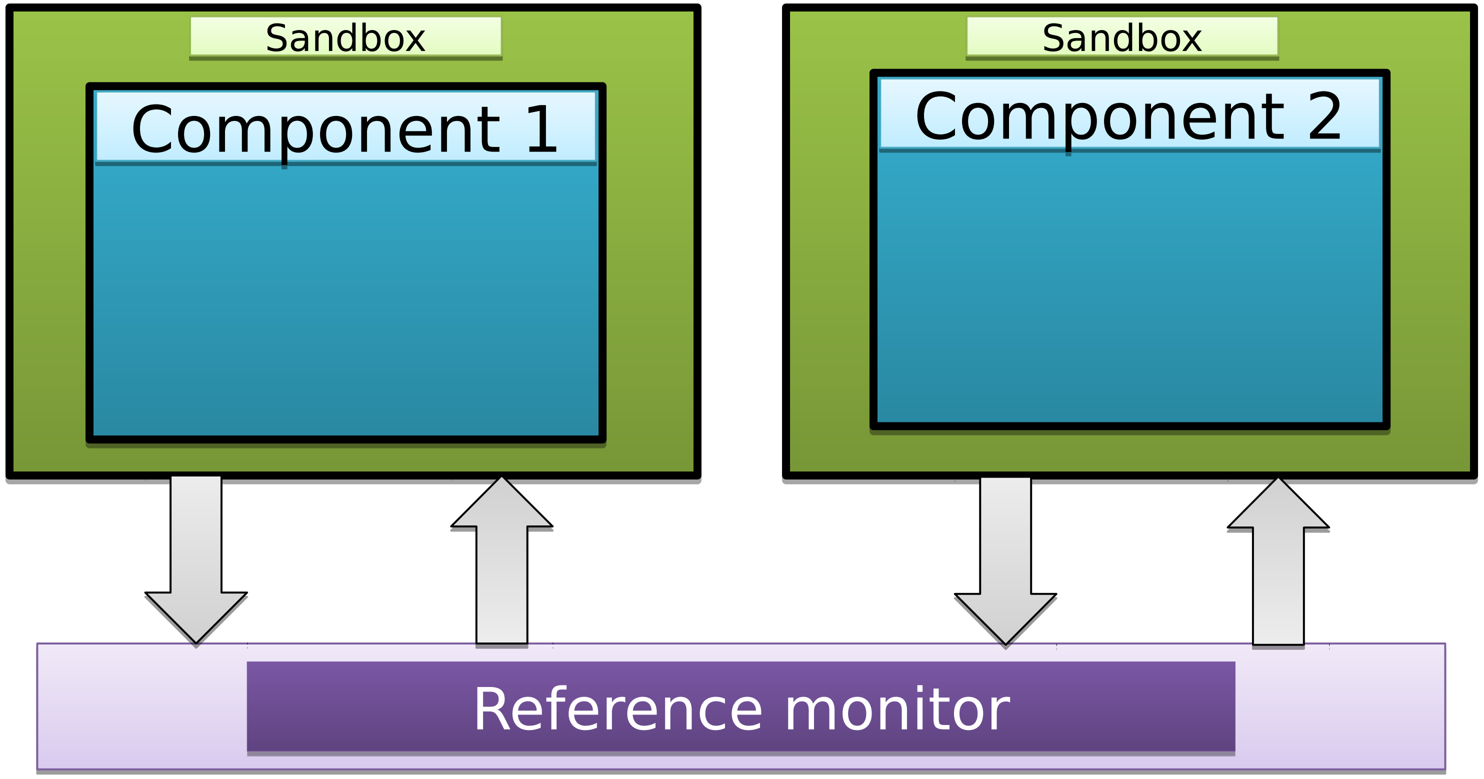
- The trusted computing base (TCB) is the subset of the system that has to be correct, for some security goal to be achieved
  - Example: the TCB for enforcing file access permissions includes the OS kernel and filesystem drivers
- TCB of the reference monitor should be small to make it verifiable

Reference Monitor and Confinement for Running Untrusted Code

We often need to run buggy/untrusted code:

- programs from untrusted Internet sites:
    - toolbars, viewers, codecs for media player
- old or insecure applications: ghostview, outlook
- legacy daemons: sendmail, bind
- Honeypots

- Goal: ensure misbehaving app cannot harm rest of system
- Approach: Confinement
    - Can be implemented at many different levels


Sandbox
Component 1
Sandbox
Component 2
Reference monitor

Confinement Examples

- Hardware: run applications on isolated hardware (air gap)
- Firewall: isolate internal network from the Internet
- Virtual machines: isolate OS's on a single machine
- Processes:
  - Isolate a process in an operating system
  - System Call Interposition

Principle of Least Privilege

- Privilege
  - Ability to access or modify a resource
- Principle of Least Privilege
  - A system module should only have the minimal privileges needed for intended purposes
- Privilege separation
  - Separate the system into independent modules
  - Each module follows the principle of least privilege
  - Limit interaction between modules

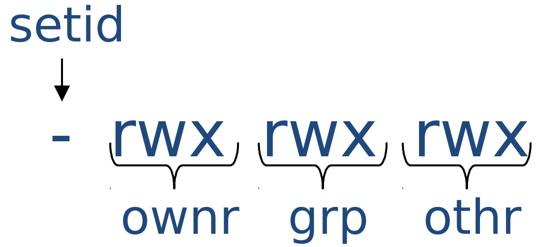
Unix access control

- File has access control list (ACL)
  - Grants permission to user ids
  - Owner, group, other
- Process has user id
  - Inherit from creating process
  - Process can change id
    - Restricted set of options
  - Special “root” id

	File 1	File 2	…
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
…			
User m	Read	write	write

Unix file access control list

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values
- Only owner, root can change permissions
  - This privilege cannot be delegated or shared
- Setid bits – Discuss in a few slides



Privileged Programs

- Privilege management is coarse-grained in today's OS
  - Root can do anything
- Many programs run as root
  - Even though they only need to perform a small number of priviledged operations
- What's the problem?
  - Privileged programs are juicy targets for attackers
  - By finding a bug in parts of the program that do not need privilege, attacker can gain root

What Can We Do?

- Drop privilege as soon as possible
- Ex: a network daemon only needs privilege to bind to low port # (<1024) at the beginning
  - Solution?
  - Drop privilege right after binding the port
- What benefit do we gain?
  - Even if attacker finds a bug in later part of the code, can’t gain privilege any more
- How to drop privilege?
  - Setuid programming in UNIX

Unix file permission

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values
- Only owner, root can change permissions
  - This privilege cannot be delegated or shared
- Setid bits

$$\underset{\downarrow}{\text{setid}}$$

$$-\ \underbrace{\text{rwx}}_{\text{ownr}}\ \underbrace{\text{rwx}}_{\text{grp}}\ \underbrace{\text{rwx}}_{\text{othr}}$$

Effective user id (EUID) in UNIX

- Each process has three Ids
  - Real user ID (RUID)
    - same as the user ID of parent (unless changed)
    - used to determine which user started the process
  - Effective user ID (EUID)
    - from set user ID bit on the file being executed, or sys call
    - determines the permissions for process
      - file access and port binding
  - Saved user ID (SUID)
    - So previous EUID can be restored
- Real group ID, effective group ID, used similarly

Operations on UIDs

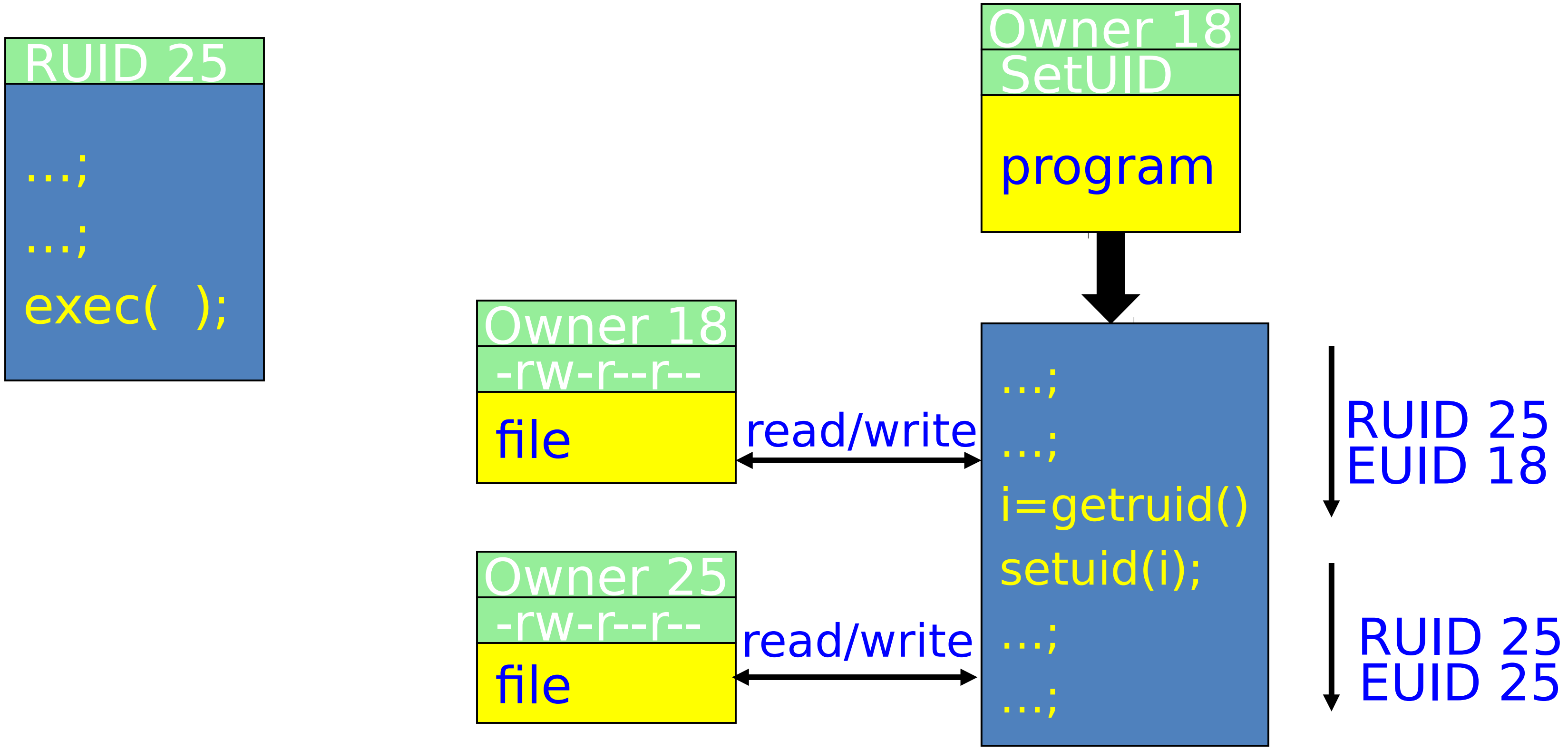
- Root
  - ID=0 for superuser root; can access any file
- Fork and Exec
  - Inherit three IDs, except exec of file with setuid bit
- Setuid system calls
  - seteuid(newid) can set EUID to
    - Real ID or saved ID, regardless of current EUID
    - Any ID, if EUID=0
- Details are actually more complicated
  - Several different calls: setuid, seteuid, setreuid

Setid bits on executable Unix file

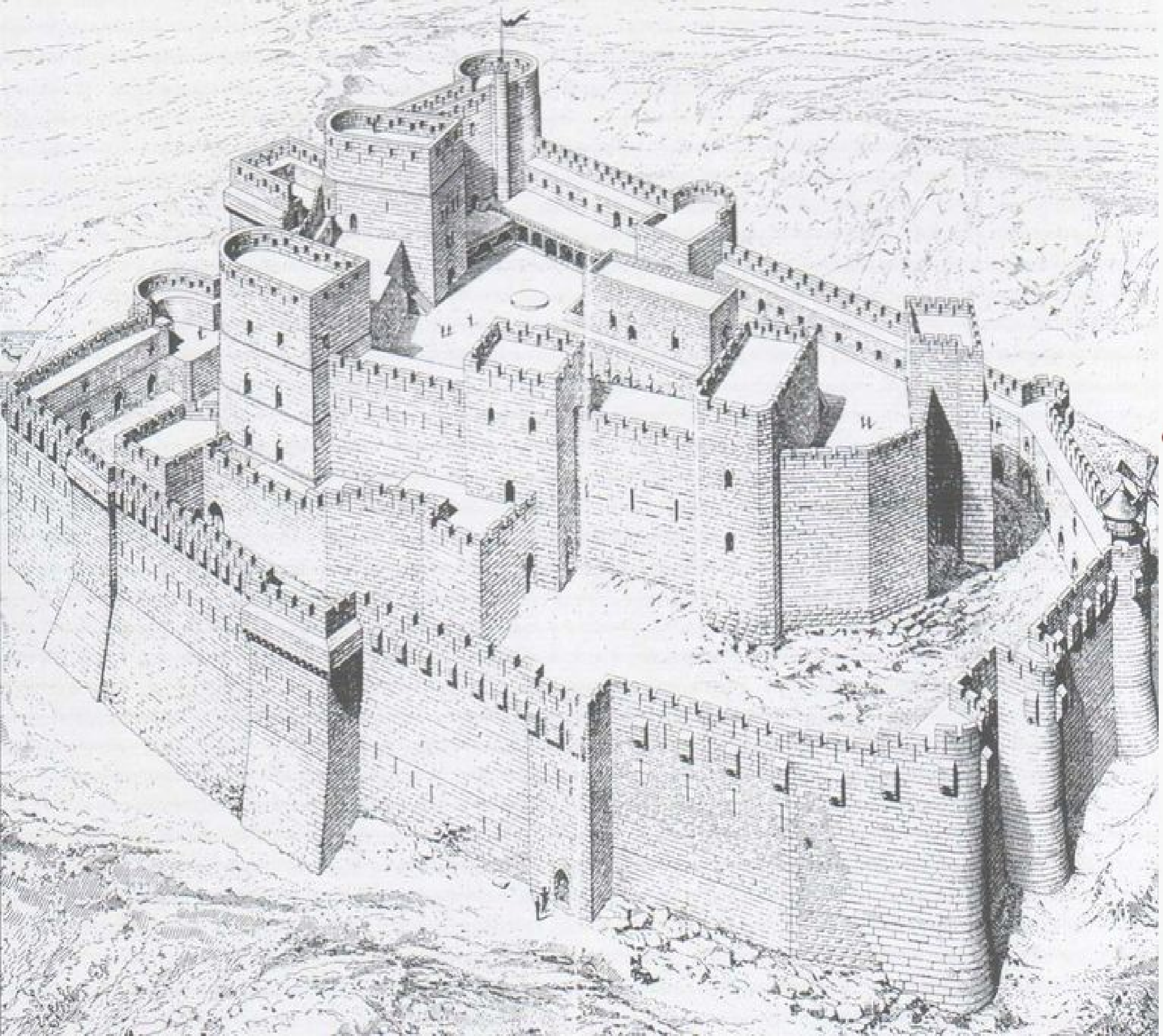
- Three setid bits
  - Setuid – set EUID of process to ID of file owner
  - Setgid – set EGID of process to GID of file
  - Sticky
    - Off: if user has write permission on directory, can rename or remove files, even if not owner
    - On: only file owner, directory owner, and root can rename or remove file in the directory

$$\begin{array}{c}\text{setid}\\ \downarrow \\ -\end{array}\ \underbrace{\text{rwx}}_{\text{ownr}}\ \underbrace{\text{rwx}}_{\text{grp}}\ \underbrace{\text{rwx}}_{\text{othr}}$$

Drop Privilege

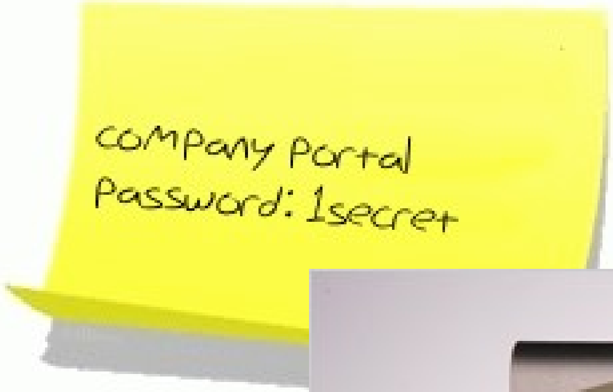


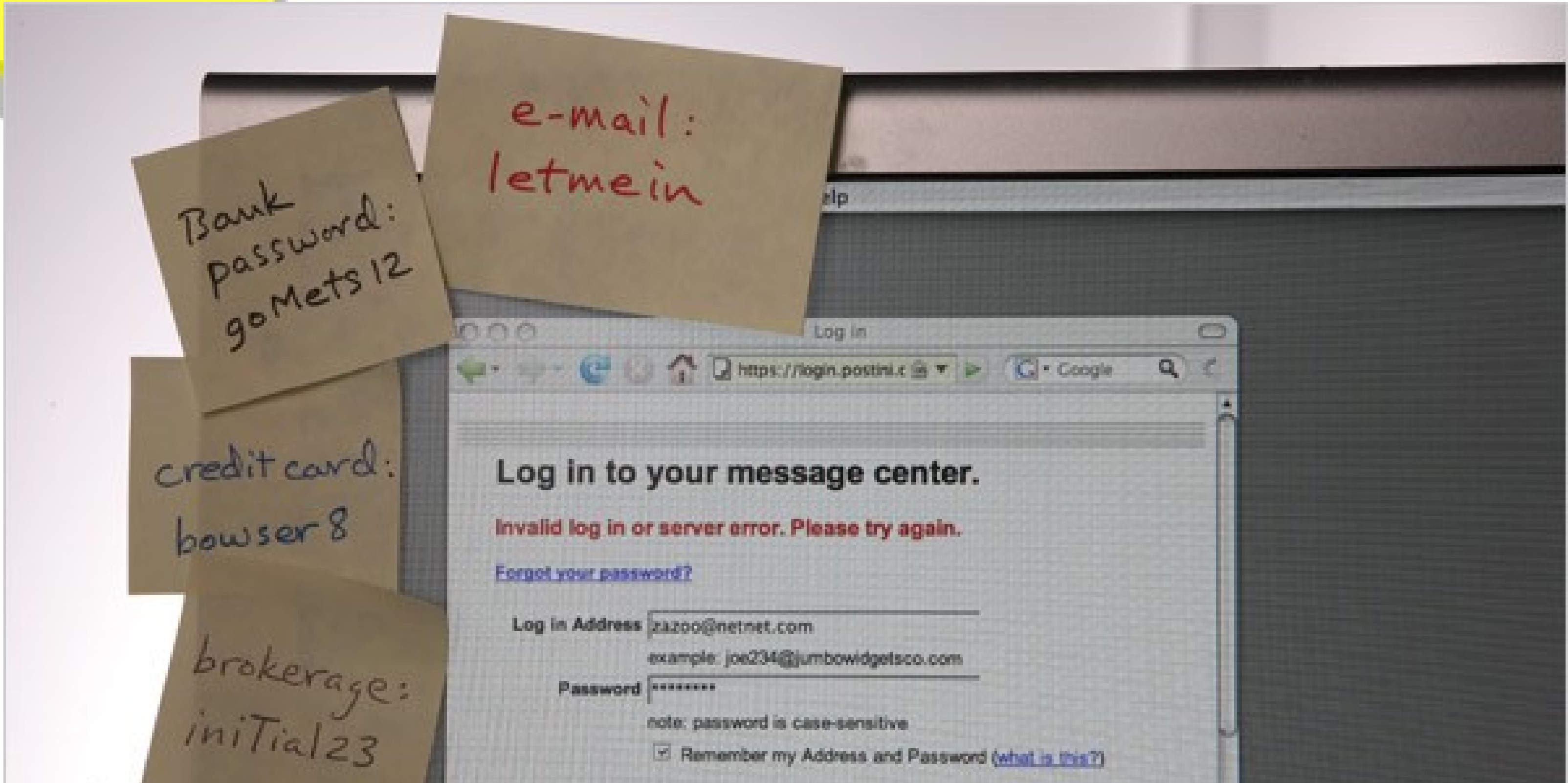
Other Security Principles



Defense in depth

- Use more than one security mechanism
- Secure the weakest link





Log in to your message center.

Invalid log in or server error. Please try again.

Forgot your password?

Log in Address zazoo@netnet.com

example: joe234@jumbowidgetsco.com

Password *********

note: password is case-sensitive

☑ Remember my Address and Password (what is this?)

“Consider human factors.”

Other Principles

- Separation of Responsibility
- “Don’t rely on security through obscurity.”
- “Fail safe.”
- “Design security in from the start.”
  - (Beware bolt-on security.)