CS165 – Computer Security

Software Vulnerabilities

January 18, 2024

Outline

- Vulnerabilities!
- Elements of a vulnerability
- Impact of vulnerability exploitation
  - Confidentiality
  - Integrity
  - Availability
- Information Flow

Vulnerability

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – A functionality that violates security
  - What violates security?
- Accessible – Adversaries may access the flaw
  - Flaw is reachable
- Exploit – Provide inputs to cause security violation
  - Adversary can produce an attack payload

E-Voting Application

- Suppose you are building an e-voting application
  - How do you ensure your application satisfies security requirements?
- What does the e-voting application do?
  - Submit a vote (by voter)
  - Tally votes (anonymized)
- What are its security requirements?
  - Let's see how we reason about security

Security Requirements

- Security requirements are described in three categories (CIA)
- Confidentiality (Secrecy)
  - Prevent leakage of sensitive data to an adversary
- Integrity
  - Prevent unauthorized modification of sensitive data
- Availability
  - Prevent blockage of use of critical services
- What security requirements should an e-voting system have?

Security Requirements of E-Voting

- Confidentiality
  - Must not release how a particular voter voted
- Integrity
  - Must not allow a voter to vote more than once
  - Each voter must vote under their own identity
- Availability
  - Must be able to tally votes
- Not an exhaustive list

Back to Flaws

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – A functionality that violates security
  - What violates a security requirement (CIA)?

Back to Flaws

- A vulnerability is a flaw (e.g., in software) that is accessible to an adversary who can exploit that flaw
- Flaw – A functionality that violates security
  - What violates a security requirement (CIA)?
- The process of voting may enable an adversary to leak another voter's vote (secrecy) or change another voter's vote (integrity)

Checking Security Requirements

- Can we reason about any of these security requirements in a systematic way?
  - To enable detection of some flaws automatically

Checking Security Requirements

- Can we reason about any of these security requirements in a systematic way?
  - To enable detection of some flaws automatically
- Answer is "Yes"
  - How?

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean vote };
File log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {   // validate voter is legit
   assign_user_vote(vote);          // get voter's vote
   log(vote, log);                  // write to vote log file
}
```

Violation of Confidentiality

- Any issues?
  - A data flow from the vote object to an external output (log operation)
    - Program does not know who can read the log file

```
log(vote, log);          // write to vote log file
```

Is an Illegal “Information Flow”

- Security requirements
  - Vote object is secret, as it must not be leaked
  - The external output is public, as it can be read outside the program
  - This data flow creates a secret → public information flow (illegal)

```
File {Public} log;

Struct Vote {Secret} vote = new Struct Vote(voter_name);
…
log(vote, log);                    // write to vote log file
```

Fix Confidentiality Violations

- How should we fix this problem?
  - In practice…
  - And with respect to information flow

Secure E-Vote Logging


```
Struct Vote { char name[LEN], Boolean vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote {Secret} vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);         // get voter's vote
   Struct EncryptedVote {Public} enc = new EncryptedVote(vote);
   log(enc, log);                   // write to vote log file
}
```

Fix Confidentiality Violation

- Solution: Write encrypted vote to log
  - Vote object is secret, as it must not be leaked
  - The external output is public, as can be read outside the program
  - But we declassify the secret by encryption, making it OK to release publicly – i.e., changing its label to public
    - Assert the result of encryption is public
  - Declassification creates a public → public information flow (legal)

Change Functionality

- Maybe we don't want to log on every vote
- For each vote of a particular type, write voter and their vote (encrypted) to the audit log
  - E.g., Write all votes since the last logging operation to a log when there is a vote for a particular candidate
    - Any issues?

```
If (vote.vote) {                         // vote.vote != 0
      Struct EncryptedVote {Public} enc = new EncryptedVote(vote);
      log(enc, log);                     // write to vote log file
}
```

Incremental E-Vote Logging

- Any issues?
  - In addition to perhaps losing some votes on a crash, one can detect which vote just happened by whether the log was written
- Consider the security requirements again
  - Encrypted vote object is public, and the external output is public
  - But, the action of writing is conditioned on a secret
    - The value of the vote
    - Why can this leak the secret value?

Incremental E-Vote Logging

- Any issues?
  - In addition to perhaps losing some votes on a crash, one can detect which vote just happened by whether the log was written
- Security requirements
  - Encrypted vote object is public, and the external output is public
  - But, the action of writing is conditioned on a secret
  - This creates a secret → public information flow (illegal)

Fix This Confidentiality Violation

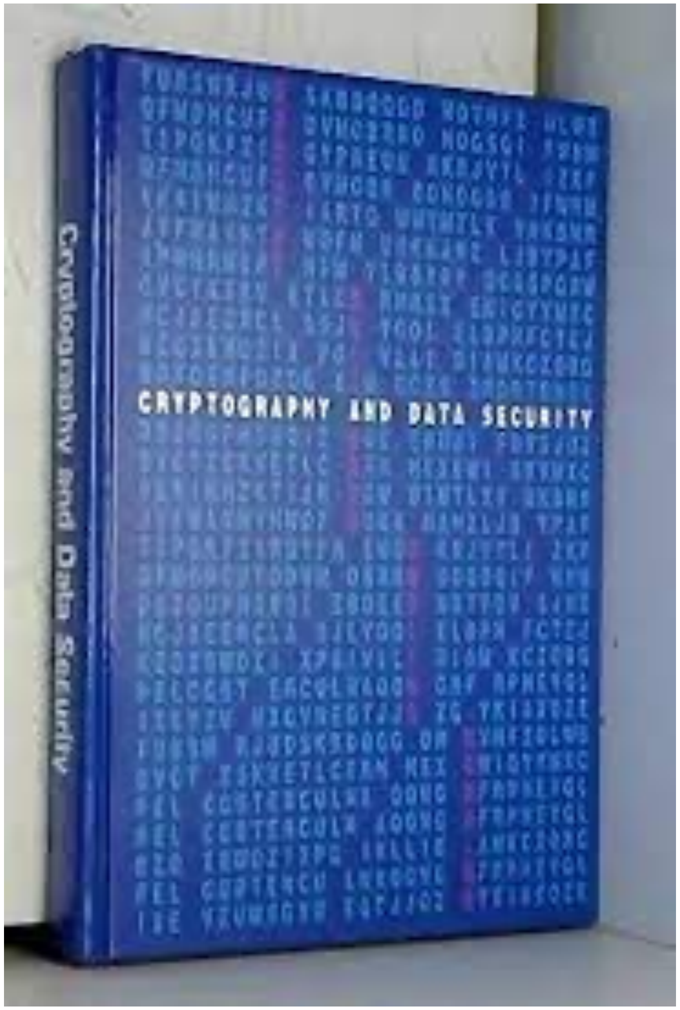
- Don’t do it
  - Do not write to public objects predicated on any secret

Explicit and Implicit Flows

- Explicit Information Flow
  - b = a
  - Explicit Information Flow: a → b
- Implicit Information Flow
  - If (a) Then b = c
  - Implicit Information Flow: a → b
- In general, we have an information flow a → b in either case

Information Flow Model

- Dorothy Denning
  - Security pioneer
- Wrote Early Security Books
  - Cryptography
  - Intrusion Detection



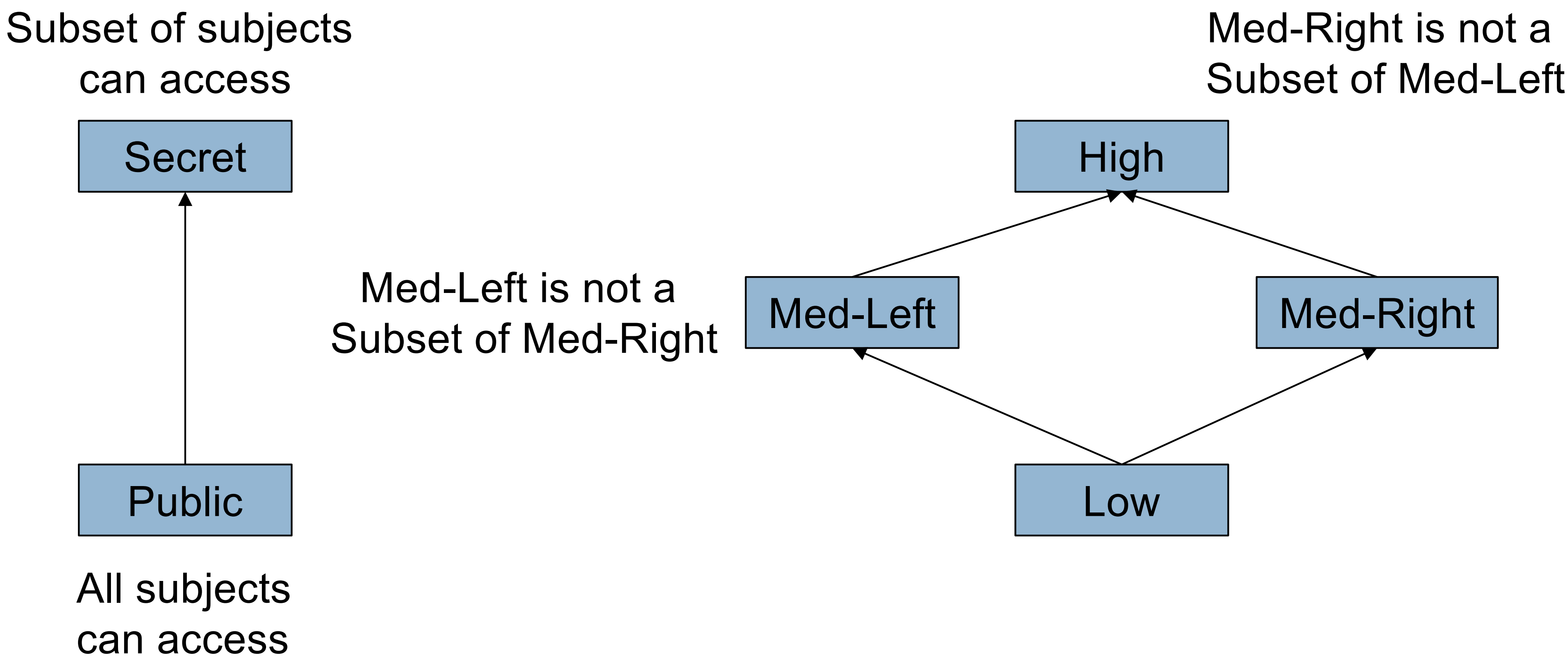
Information Flow Model

- A program consists of (recursively)
  - An elementary statement – $S = S_i$
  - A sequence of elementary statements – $S = S_1; \ldots; S_n$
  - A conditional statement – $c: S_1; \ldots; S_n$
    - A set of sequences conditioned based on the value of c
- Statements may create explicit or implicit flows
  - Implicit flows can only be the result of a conditional
- Goal: all explicit and implicit flows are "secure"
  - What does security mean?

Lattice Security Model

- Formalizes security based on information flow models
  - FM = {N, P, SC, /, >}
- Information flow model instances form a lattice
  - What's a lattice?
    - Graph where every node has a LUB and a GLB
- N are objects, P are processes, and each are assigned a security class SC
  - {SC, >} is a partial ordered set
  - SC, the set of security classes, is finite
  - SC has a lower bound
  - and / is a LUB operator

Lattice Examples

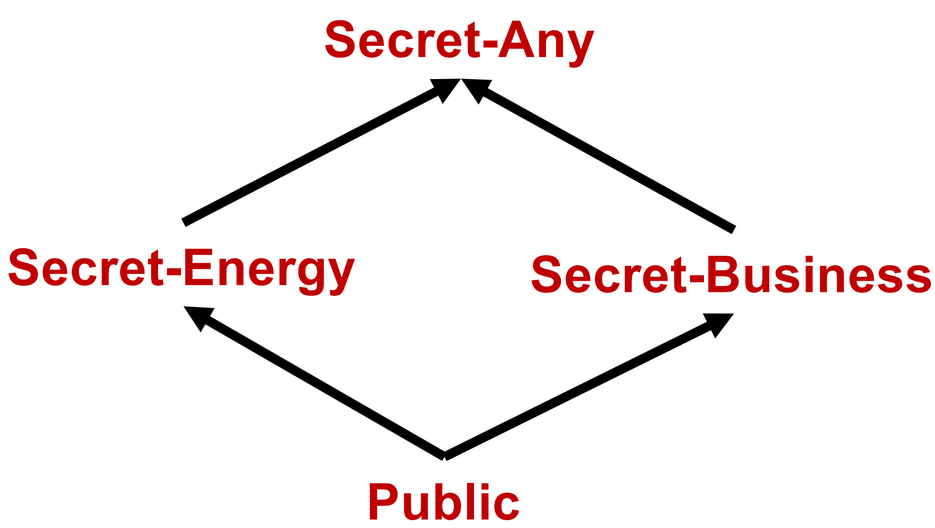


Simple Security Lattice Example

- You have N objects and P processes
  - Each is assigned to a security class in SC
  - Where SC = {Public, Secret}
- {SC, >} forms a partially ordered set where
  - Secret > Public
    - Meaning data from Public objects/processes can flow to Secret objects/processes, but not vice versa
  - What does this security requirement represent?
- SC has a lower bound (Public)
- / is a LUB operator
  - Represents what happens when two objects are combined
  - Secret / Public ➔ ???

Complex Security Lattice Example

- You have N objects and P processes
  - Each is assigned to a security class in SC
  - Where SC = {Public, Secret-Energy, Secret-Business, Secret-Any}



Complex Security Lattice Example

- You have N objects and P processes
  - Each is assigned to a security class in SC
  - Where SC = {Public, Secret-Energy, Secret-Business, Secret-Any}
- {SC, >} forms a partially ordered set where
  - What information flows are allowed here?
- SC has a lower bound (Public)
- / is a LUB operator
  - Secret-Energy / Public → ???
  - Secret-Energy / Secret-Business → ???

Complex Security Lattice Example

- You have N objects and P processes
  - Each is assigned to a security class in SC
  - Where SC = {Public, Secret-Energy, Secret-Business, Secret-Any}
- {SC, >} forms a partially ordered set where
  - What information flows are allowed here?
- SC has a lower bound (Public)
- / is a LUB operator
  - Secret-Energy / Public → Secret-Energy
  - Secret-Energy / Secret-Business → Secret-Any

What Is This Good For?

What Is This Good For?

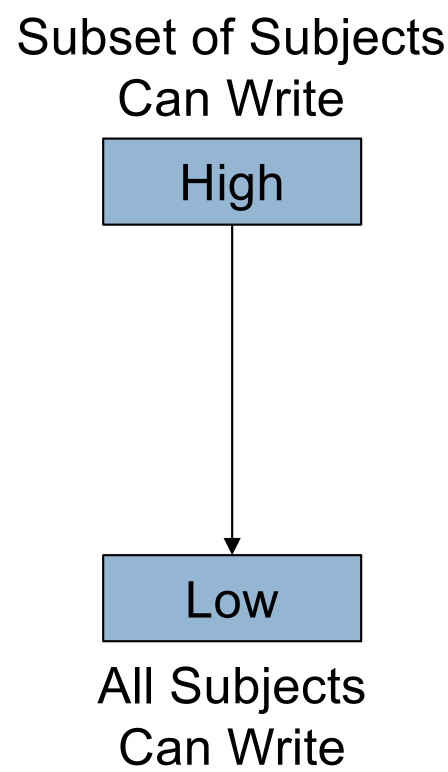
- Let’s Find Some Vulnerabilities!

Integrity Lattice

- We have mainly used information flow to find vulnerabilities that violate integrity
- Security classes for integrity
  - SC = {High, Low}
- {SC, >} forms a partially ordered set where
  - What information flows are allowed here?
- SC has a lower bound (Low) and / is a LUB op
  - Reverse legal information flows (“no write up”)

Integrity Lattice

- We have mainly used information flow to find vulnerabilities that violate integrity
- Security classes for integrity
  - SC = {High, Low}
- {SC, >} forms a partially ordered set where
  - What information flows are allowed here?
- SC has a lower bound (High) and / is a LUB op
  - Reverse legal information flows ("no write up")



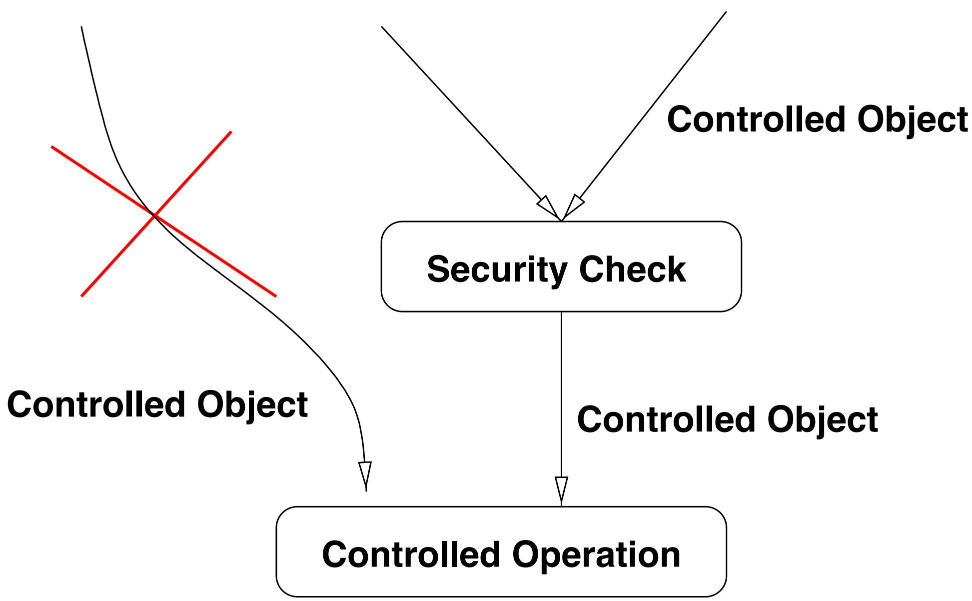
Linux Access Control

- Linux introduced checks to enforce access control
  - Called the Linux Security Modules
- Idea: Check the access control policy before each "security-sensitive-operation" made by the Linux kernel
- How do we know that all security-sensitive operations (e.g., llseek) are checked correctly?

```
/* Code from fs/read_write.c */
sys_lseek(unsigned int fd, ...)
{
  struct file * file;
  ...
  file = fget(fd);
  retval = security_ops->file_ops
             ->llseek(file);
  if (retval) {
    /* failed check, exit */
    goto bad;
  }
  /* passed check, perform operation */
  retval = llseek(file, ...);
  ...
}
```

Complete Mediation

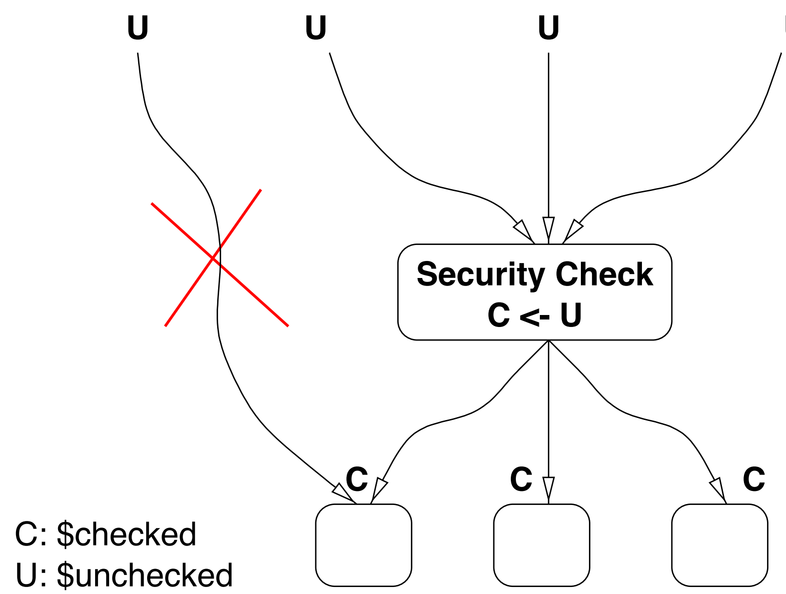
- All security-sensitive operations on an object must be preceded by an access control check on that object



- How do we use information flow to validate complete mediation and find vulnerabilities?

Complete Mediation

- All system calls produce an “unchecked” object (low)
- All checks declassify an “unchecked” object to a “checked” object (high)
- Every security-sensitive operation must be performed on a ”checked” object (high)



Vulnerability Found

- Found several vulnerabilities in Linux Security Modules
- These were fixed prior to upstreaming, providing confidence in the implementation
- One example: Found the presence of a check, but not on the object used in the security-sensitive operation

```
/* from fs/fcntl.c */
long sys_fcntl(unsigned int fd,
               unsigned int cmd,
               unsigned long arg)
{
  struct file * filp;
  ...
  filp = fget(fd);
  ...

  err = security_ops->file_ops
        ->fcntl(filp, cmd, arg);
  ...
  err = do_fcntl(fd, cmd, arg, filp);

  ...
}

static long
do_fcntl(unsigned int fd,
         unsigned int cmd,
         unsigned long arg,
         struct file * filp) {
  ...
  switch(cmd){
    ...
    case F_SETLK:

      err = fcntl_setlk(fd, ...);

    ...
  }
  ...
}

/* from fs/locks.c */
fcntl_getlk(fd, ...) {
  struct file * filp;
  ...

  filp = fget(fd);

  /* operate on filp */
  ...
}
```

Conclusions

- Vulnerabilities that compromise confidentiality or integrity are common
- Theory: Program information flows (according to Denning's Information Flow Model) must comply with confidentiality and integrity (as defined by Denning's Lattice Security Model)
- Can be used to find real vulnerabilities
- We are still building tools that leverage this approach today – albeit augmented

Questions

Example

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean vote };
File log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {   // validate voter is legit
   assign_user_vote(vote);          // get voter’s vote
   log(vote, log);                  // write to vote log file
}
```

E-Vote Logging

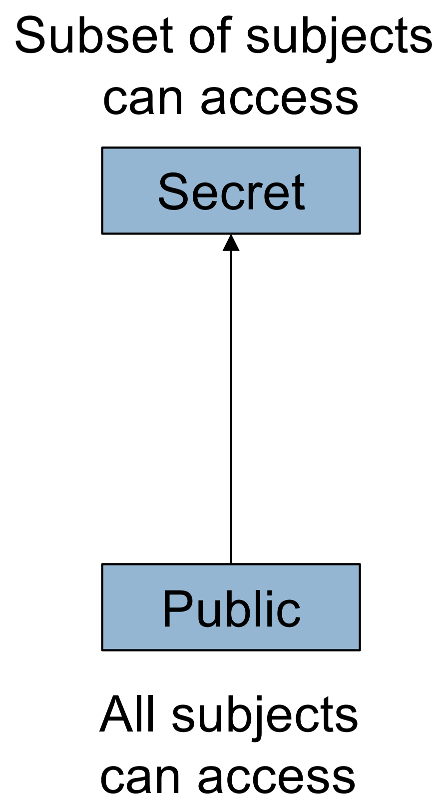

```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {   // validate voter is legit
   assign_user_vote(vote);          // get voter’s vote
   log(vote, log);                  // write to vote log file
}
```

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {   // validate voter is legit
   assign_user_vote(vote);          // get voter's vote
   log(vote, log);                  // write to vote log file
}
```

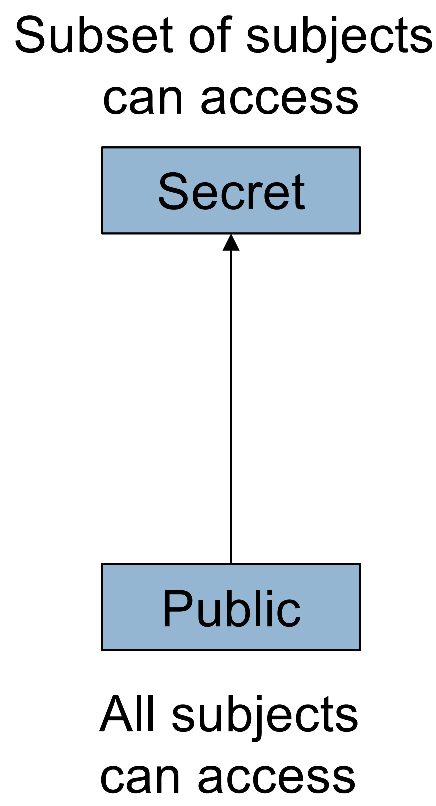


E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);         // get voter's vote
   log(vote, log);                 // write to vote log file
}
```

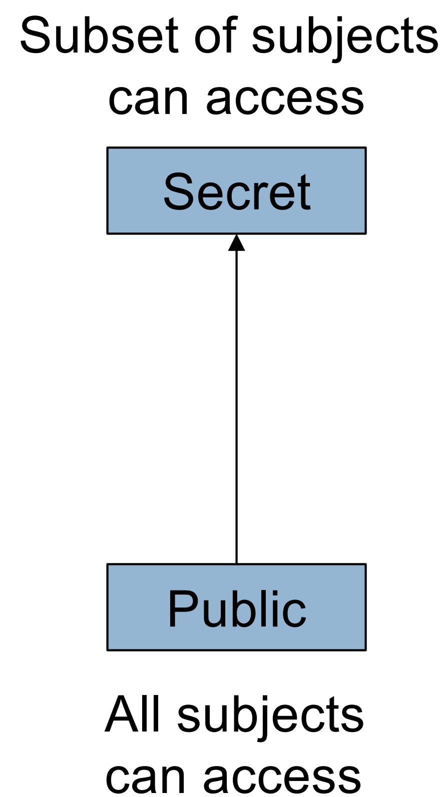
No label for `voter_name`



E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive vote request(voter name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);         // get voter’s vote
   log(vote, log);                 // write to vote log file
}
```

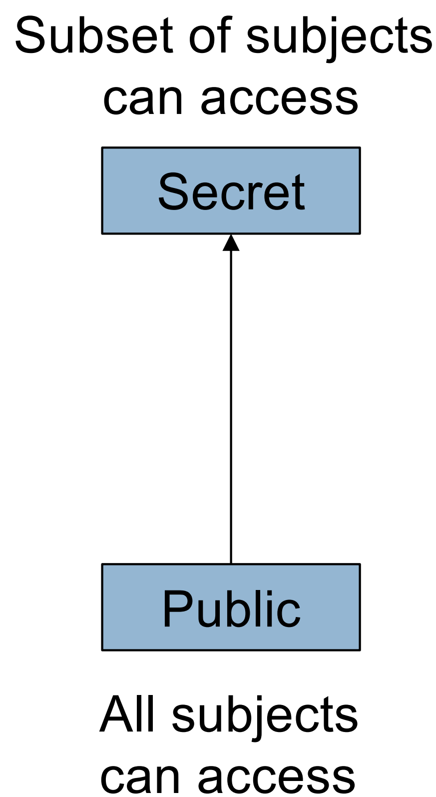


`vote.vote` is set to null and Secret
`vote.name` is unlabeled

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);          // get voter's vote
   log(vote, log);                  // write to vote log file
}
```



Suppose `authenticate(vote)` only returns whether the user of `vote.name` authenticated. This is public knowledge, so declassified to {Public}

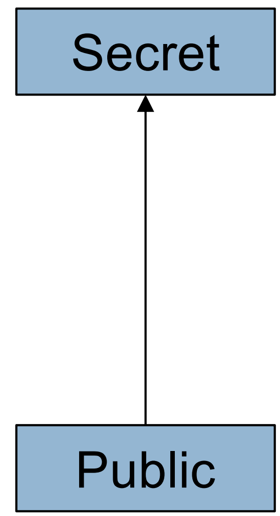
E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
    assign_user_vote(vote);        // get voter's vote
    log(vote, log);                // write to vote log file
}
```

`Vote.vote` is updated, which is {Secret}

Subset of subjects can access

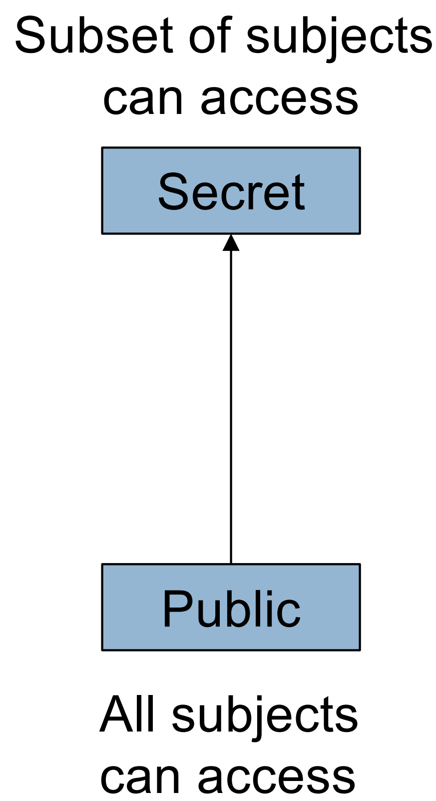


All subjects can access

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);         // get voter's vote
   log(vote, log);                 // write to vote log file
}
```



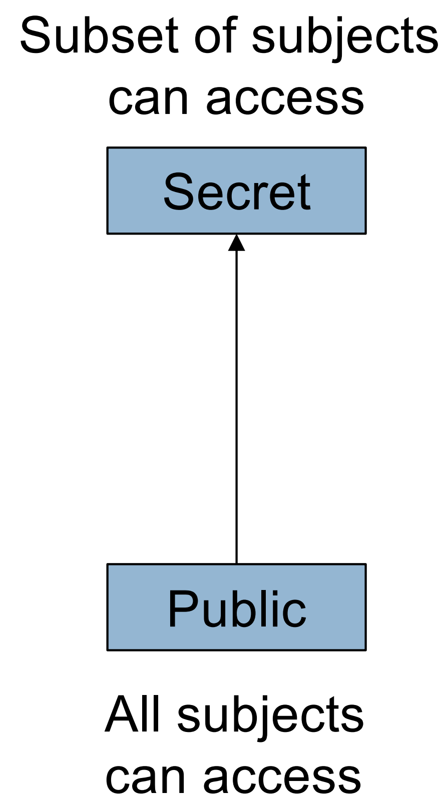
`Vote.vote` is recorded in `log`, which creates a {Secret} → {Public} flow. Illegal

E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
   assign_user_vote(vote);         // get voter's vote
   log(vote, log);                 // write to vote log file
}
```

What about this implicit flow?

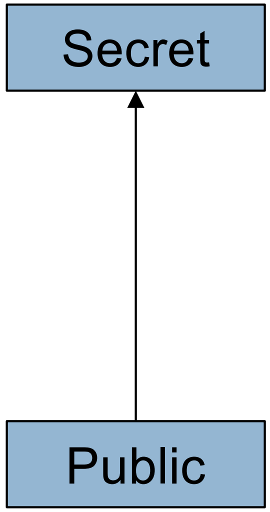


E-Vote Logging


```
Struct Vote { char name[LEN], Boolean {Secret} vote };
File {Public} log;

Loop:
Receive_vote_request(voter_name);
Struct Vote vote = new Struct Vote(voter_name);
If (authenticate(vote) == TRUE) {  // validate voter is legit
    assign_user_vote(vote);          // get voter's vote
    log(vote, log);                  // write to vote log file
}
```

Subset of subjects can access



All subjects can access

What about this implicit flow?
Creates a {Public} → {Public} flow. Legal