CMPSC 497
Info Flow Defenses

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Vulnerabilities (attack vectors)

• A **vulnerability** is a **flaw** that is **accessible** to an **adversary** who can **exploit** that flaw

• E.g., buffer overflow, file open w/ adversary name

• What is the source of a vulnerability?
  ‣ Bad software (or hardware)
  ‣ Bad design, requirements
  ‣ Bad policy/configuration
  ‣ System Misuse
  ‣ Unintended purpose or environment
    • E.g., student IDs for liquor store
Vulnerability Exploits

- Must violate some security requirements – by exploiting a vulnerability
  - Secrecy
    - Leak sensitive data to adversary
  - Integrity
    - Allow adversary to modify unauthorized data
  - Availability
    - Allow adversary to block functionality
- We’ll focus on first two categories today
Prevent Vulnerabilities

- Is it possible to prevent vulnerabilities that violate secrecy and integrity in a systematic manner?
Can You Make Secure?

- Can you fix the program to be secure?

```java
if (password_hash == hash(password))
    return authenticated;
else
    return !authenticated;
...
if (authenticated)
    return “You are authenticated”
```

- Need to check the password and the program behavior depends on that
Can Prove It Secure?

• Can you fix the program to be secure?

```python
if (password_hash == hash(password))
    return authenticated;
else
    return !authenticated;
...
if (authenticated)
    return "You are authenticated"
```

• Need to check the password and the program behavior depends on that
Implicit and explicit flows

• Explicit
  ‣ Direct transfer to b from a (e.g., b = a)

• Implicit
  ‣ Where value of b may depend on value of a indirectly (e.g., if a = 0, then b = c)

• Model covers all programs
  ‣ Statement S
  ‣ Sequence S1, S2
  ‣ Conditional c: S1, ..., Sm

• Implicit flows only occur in conditionals
Semantics

- Program is secure if:
  - Explicit flow from S is secure
  - Explicit flow of all statements in a sequence are secure (e.g., S1; S2)
  - Conditional \( c: S_1, \ldots, S_m \) is secure if:
    - The explicit flows of all statements \( S_1, \ldots, S_m \) are secure
    - The implicit flows between c and the objects in Si are secure
Noninterference

- These security requirements imply a property called noninterference

- Intuitively, noninterference implies that a public subject cannot see the effect of updates to secret memory
  - No explicit flows – cannot see secrets directly
  - No implicit flows – cannot see public data whose values may be influenced by secret data

- Let’s see what this means
Noninterference

- Suppose we have a program that takes high inputs and low inputs and produces some observable outputs.
- A system satisfies noninterference if any variation of its high inputs causes no difference in its observable outputs.
Noninterference

- A memory $M$ consists of two projections $M_L$ and $M_H$ for low and high memory, respectively.
- Program operations alter memory $(P, M) \rightarrow M'$.
- Also there is a function $M =_L M'$ that compares low memories.
- A system satisfies noninterference iff for all memories $M_1$ and $M_2$:
  - $M_1 =_L M_2$, $(P, M_1) \rightarrow M_1'$, $(P, M_2) \rightarrow M_2'$, $M_1' =_L M_2'$.
- Meaning that the observable (low) memory is the same for any high inputs.
Implicit Flows

• Is this program secure?

    do {  // password hash is secret
        if ( password_hash == hash(password) )
            return authenticated;  // secret: impl
    
        else {
            return !authenticated; // secret: impl
            try++;
        }
    }
Information Flow Security

- Check that your program obeys information flow security at compile time
  - What do we need to do?
- We need
  - Program
  - Information Flow Policy
  - Mapping between the policy and the program
    - Annotation
  - Policy verification for the annotated program
Denning’s Lattice Policy

- Formalizes information flow models
  - \( FM = \{N, P, SC, /, >\} \)
- Shows that the information flow policies form a lattice
  - \( N \) are objects, \( P \) are processes,
  - \( \{SC, >\} \) is a partial ordered set,
  - \( SC \), the set of security classes is finite,
  - \( SC \) has a lower bound,
  - and \( / \) is a lub operator
- A **lattice** is a partially ordered set where each pair of elements of the sets has a unique least upper bound (LUB) and a unique greatest lower bound (GLB)
  - Why do you think that is desirable?
Suppose we have the following code

```c
int {secret} a = 1;
int {public} b = 2;
c = a + b;
```

What is the label of c?
Lattice

• Suppose we have the following code

```c
int {secret} a = 1;
int {public} b = 2;
c = a + b;
```

• What is the label of c?

• The lattice policy has two labels (security classes) `public` and `secret`, where `secret > public`
  ‣ We use the LUB (join) to determine the label of the combined data
Lattice

• Suppose we have the following code

```c
int {A} a = 1;
int {B} b = 2;
c = a + b;
```

• What is the label of `c`?

• The lattice policy has four labels (security classes) `A`, `B`, and `{A, B}`, and `{}`
  - We use the LUB (join) to determine the label of the combined data
Mapping

- Programmers choose the labels to assign their data

```c
int {A} a = 1;
int {B} b = 2;
c = a + b;
```

- But, you need not assign a label to every variable as we saw for ‘c’
Static and Dynamic Binding

• Static binding
  ‣ Security class of an object is assigned manually
  ‣ The security class does not change

• Dynamic binding
  ‣ Security class of an object is assigned based on its use
  ‣ For \( c = b + a \), then the security class of \( c \) is \( b / a \)
Implicit Flows

- Implicit flows are dynamically bound

```java
do {  // password_hash is statically labeled secret
    if ( password_hash == hash(password) )
        return authenticated;  // secret: impl
    else {
        return !authenticated; // secret: impl
        try++;
    }
}
```
Policy Verification

- Check Denning’s rules for information flow
- By propagating labels among variables
- As long as a variable
  - Only appears in statements that comply with its static labeling
  - OR
  - Can be assigned a label dynamically that satisfies the constraints implied by prior label assignments
- The program is enforces the information flow policy
Policy Verification

- How do we check information flow efficiently for programs with multiple functions?
  - Can we check each function independently?
Type Safety

- A type-safe language maintains the semantics of types. E.g. can’t add int’s to Object’s.

- Type-safety is compositional. A function promises to maintain type safety.

**Example 1**
Object obj;
int i;
obj = obj \times i;

**Example 2**
String proc_obj(Object o);
...
main()
{
    Object obj;
    String s = proc_obj(obj);
    ...
}
Security Types

Example 1
```java
int{high} h1, h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
X h2 + l;
```

- Key insight: label types with security levels
- Security-typing is compositional

Example 2
```java
String{low}
proc(Object{high} o);
...
main()
{
    Object{high} obj;
    String{low} s;
    s = proc_obj(obj);
    ...
}
```
Composition

• Types for return of check_passwd and &try

```c
foo() {
...
    rtn = check_passwd( username, password, &try );
    ...}

int check_passwd( char *uname, char *passwd, int *try ) {
    Hash password_hash = lookup_hash( uname );
    if ( password_hash == hash(passwd) )
        return {public} authenticated;
    else {
        try++;
        return {public} !authenticated;
    }
}
```
Declassification

- One may need to release data from a process that may be affected by secrets

- **Declassification** methods aims to remove secrets from data prior to release
  - How would we declassify medical records?
  - How would we declassify password verification responses?

- A password checking limit is a form of a declassifier, as it allows the leakage of some limited amount of secret information
• Is this program secure?

```c
do {  // password hash is secret
    if ( password_hash == hash(password) )
        return authenticated;  // secret: impl
    else {
        return !authenticated; // secret: impl
        try++;
    }
} 
while ( try < limit );  // declass: authenticated
```
• Is this program secure?

    do {  // password hash is secret
        if ( password_hash == hash(password) )
            return {public} authenticated;
        else {
            *try = {public} (*try+1);
            return {public} !authenticated;
        }
    } 
while ( try < limit );
Declassification

- As you can see declassification is asserted in programming languages that are information-flow aware
  - May not be correctly declassified
  - Open problem to verify
    - Because declassification is often application specific
Java Information Flow

- Java language extended with security labels in the type qualifiers
  - Security-type system
  - Rich label model for principals – outside scope
- Enforce information flow at compile time
  - And at runtime with “runtime types,” which are outside our scope today
- Language is called Jif and examples are based on the language
Jif Applications

- Jif has been used to deploy applications
  - Secure voting
  - Mail client (PSU)
Take Away

- **Vulnerabilities** are possible when an adversary exploits information flows.
- Enforce **information flow** which prevent leakage of secrets and unauthorized modification of high integrity data.
- Programming language features enable checking of information flow policies:
  - Noninterference
- Fixing information flow errors in a secure way is hard.