Module: Future of Secure Programming

Professor Trent Jaeger
PENN STATE UNIVERSITY
Programmer’s Problem

• Implement a program
  ‣ Without creating vulnerabilities

• What is a vulnerability?
Software Vulnerabilities

• Vulnerability combines
  ‣ A flaw
  ‣ Accessible to an adversary
  ‣ Who can exploit that flaw

• Which would you focus on to prevent vulnerabilities?
Software Vulnerabilities

• Some examples
  ‣ Buffer overflow
  ‣ Link traversal
  ‣ Authorization bypass
Buffer Overflow Static Analysis

• For C code where
  ‣ char dest[LEN]; int n;
  ‣ ...
  ‣ n = input();
  ‣ ...
  ‣ strncpy(dest, src, n);

• Can this code cause a buffer overflow?
Runtime Analysis

• One approach is to run the program to determine how it behaves

• Analysis Inputs
  ‣ **Input Values** - command line arguments
  ‣ **Environment** - state of file system, environment variables, etc.

• Question
  ‣ Can any input value in any environment cause a flaw (e.g., buffer overflow)?

• What are limitations of runtime analysis?
Static Analysis

• Explore all possible executions of a program
  ‣ All possible inputs
  ‣ All possible states
Static Analysis

• Provides an approximation of behavior
• "Run in the aggregate"
  ‣ Rather than executing on ordinary states
  ‣ Finite-sized descriptors representing a collection of states
• "Run in non-standard way"
  ‣ Run in fragments
  ‣ Stitch them together to cover all paths
• Runtime testing is inherently incomplete, but static analysis can cover all paths
Static Analysis Example

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For an expression, \( c = a \times b \)
  - If \( a \) has a descriptor \( pos \)
  - And \( b \) has a descriptor \( neg \)
- What is the descriptor for \( c \) after that instruction?
- How might this help?
Descriptors

- Choose a set of descriptors that
  - Abstracts away details to make analysis tractable
  - Preserves enough information that key properties hold
  - Can determine interesting results

- Using sign as a descriptor
  - Abstracts away specific integer values (billions to four)
  - Guarantees when $a \times b = 0$ it will be zero in all executions

- Choosing descriptors is one key step in static analysis
Buffer Overflow Static Analysis

• For C code where
  ‣ char dest[LEN]; int n;
  ‣ n = input();
  ‣ strncpy(dest, src, n);

• Static analysis will try all paths of the program that impact variable n and flow to strncpy
  ‣ May be complex in general because
    • **Paths**: Exponential number of program paths
    • **Interprocedural**: n may be assigned in another function
    • **Aliasing**: n’s memory may be accessed from many places

• What **descriptor values** do you care about for n?
Remove All Flaws

• Can we remove all flaws from software?

• Consider buffer overflows
  ‣ Suppose you find code that allows writing beyond the end of a buffer
  ‣ Should any code that permits buffer overflow be fixed?

• Challenge: Identifying all buffer overflows is difficult
  ‣ Problem: Alias Analysis
    • Technique to determine if a storage location may be accessed in multiple ways
Limitations of Static Analysis

- **Scalability**
  - Can be expensive to reason about all executions of complex programs

- **False positives**
  - Overapproximation means that executions that are not really possible will be modeled

- **Accuracy**
  - Alias analysis and other imprecision may lead to false negatives
  - Sound methods (no false negatives) can exacerbate scalability and false positives problems

- **Bottom line**: Static analysis often must be directed
Other Solutions

- Don’t use languages that permit buffer overflows

- **Type Safety**
  - Program operations must comply with semantics of types being operated upon
    - E.g., cannot write beyond the end of a buffer
    - E.g., cannot use a reference of one type as a reference to an incompatible type (pointers)
  - Many languages have type-safe properties, although full type-safety requires proofs using precise program semantics
    - ML has been shown to be type-safe, as has a subset of Haskell
    - Java has some type-safe properties (modulo runtime errors)
Resource Access Vulnerabilities

• What about other types of vulnerabilities?
  ‣ **Link traversal**

• See the open on line 6
  ‣ How do we know we received the right resource?
  ‣ Adversary can redirect us to another resource of her choice using links

• How do we prevent such vulnerabilities?

```c
/* fail if file is a symbolic link */
int open_no_symlink(char *fname)
{
    struct stat lbuf, buf;
    int fd = 0;
    lstat(fname, &lbuf);
    if (S_ISLNK(lbuf.st_mode))
        error("File is a symbolic link!");
    fd = open(fname);
    fstat(fd, &buf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Race detected!");
    lstat(fname, &buf);
    if ((buf.st_dev != lbuf.st_dev) ||
        (buf.st_ino != lbuf.st_ino))
        error("Cryogenic sleep race!");
    return fd;
}
```
Preventing Resource Access

• Two ways
  ‣ Extended system call API
    • Check for links and check properties of resource (lstat, fstat)
  ‣ Access control policy
    • Prevent adversary access to resource retrieval

• Problem: Programmer only knows about the former
  ‣ Does every open not protected by lstat and fstat lead to a vulnerability?
Preventing These Vulnerabilities

• What can the programmer do to secure their program in such cases?
Information Flow Control

Access to all compartments

{NUC, EUR, US}

{NUC, EUR} {NUC, US} {EUR, US}

Access to no compartments

{NUC} {EUR} {US}

∅
Information Flow Control

• What is it?
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

Access to all compartments
{NUC, EUR, US}

{NUC, EUR} {NUC, US} {EUR, US}

Access to no compartments
{NUC} {EUR} {US} Ø
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

• Why?
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

• Why?
  ‣ Edward Snowden, e.g.
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

• Why?
  ‣ Edward Snowden, e.g.

• Problem: Information release
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

• Why?
  ‣ Edward Snowden, e.g.

• Problem: Information release

• Solution: Information Flow Control
Information Flow Control

• What is it?
  ‣ Simple security & ★-property

• Why?
  ‣ Edward Snowden, e.g.

• Problem: Information release

• Solution: Information Flow Control

• Actually stronger enforcement than MLS
Denning’s Lattice Model

- Formalizes information flow models
  - \( FM = \{N, P, SC, \leq, \geq\} \)

- Shows that the information flow model instances form a lattice
  - \( N \) are objects, \( P \) are processes,
  - \( \{SC, \geq\} \) is a partial ordered set,
  - \( SC \), the set of security classes is finite,
  - \( SC \) has a lower bound,
  - and \( \leq \) is a lub operator

- Implicit and explicit information flows

- Semantics for verifying that a configuration is secure

- Static and dynamic binding considered

- Biba and BLP are among the simplest models of this type
Denning’s Lattice Model

- Formalizes information flow models
  - \( FM = \{N, P, SC, /, >\} \)

- Shows that the information flow model instances 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

- Implicit and explicit information flows
- Semantics for verifying that a configuration is secure
- Static and dynamic binding considered
- Biba and BLP are among the simplest models of this type
Denning’s Lattice Model

• Formalizes information flow models
  ‣ $\text{FM} = \{N, P, SC, /, >\}$

• Shows that the information flow model instances 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

• Implicit and explicit information flows

• Semantics for verifying that a configuration is secure

• Static and dynamic binding considered

• Biba and BLP are among the simplest models of this type
Denning’s Lattice Model

• Formalizes information flow models
  ‣ \( FM = \{N, P, SC, /, >\} \)

• Shows that the information flow model instances 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

• Implicit and explicit information flows

• Semantics for verifying that a configuration is secure

• Static and dynamic binding considered

• Biba and BLP are among the simplest models of this type
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**
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**
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
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: S1, ..., Sm is secure if:
    • The explicit flows of all statements S1, ..., Sm are secure
    • The implicit flows between c and the objects in Si are secure
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: S1, …, Sm is secure if:
    - The explicit flows of all statements S1, …, Sm are secure
    - The implicit flows between c and the objects in Si are secure
Semantics

- Program is secure if:
  - Explicit flow from \( S \) is secure
  - Explicit flow of all statements in a sequence are secure (e.g., \( S_1; S_2 \))
  - 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 \( S_i \) are secure
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: S1, ..., Sm is secure if:
    • The explicit flows of all statements S1, ..., Sm are secure
    • The implicit flows between c and the objects in Si are secure
Build on Type Safety
Example 1
Object obj;
int i;
obj = obj + i;
Build on Type Safety

Example 1
Object obj;
int i;
obj = obj + i;
Build on Type Safety

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

Example 1
Object obj;
int i;
obj = obj + i;
Build on 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 + i;  // Red X
Build on 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
```java
Object obj;
int i;
obj = obj + i;
```

Example 2
```java
String proc_obj(Object o);
...

main()
{
    Object obj;
    String s = proc_obj(obj);
    ...
}
```
Labeling Types
Labeling Types

Example 1
int{high} h1, h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
l = h2 + l;
Labeling Types

Example 1

```c
int{high} h1, h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
\[\times\] l = h2 + l;
```

\[\times\] indicates an error in the code.
Labeling Types

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

- **Key insight:**
  label types with security levels
Labeling Types

Example 1
int{high} h1,h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
l \times h2 + l;

• **Key insight:**
  label types with security levels

• **Security-typing is compositional**
Labeling Types

Example 1
int{high} h1, h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
\[\text{X} \quad l = h2 + l;\]

• Key insight:
  label types with security levels

• Security-typing is compositional

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

**Implicit Flows**

Static (virtual) tagging

```c
int mydata = 0;
int mydata2 = 0;
if (test)
  mydata = 1;
else
  mydata = 2;
mydata2 = 0;
print(mydata2);
print(mydata);
```

*mydata contains information about test so it can no longer be Low, but mydata2 is outside the conditional, so it is untainted by test*

Causes type error at compile-time
Implicit Flows

Static (virtual) tagging

```
int mydata = 0;
int mydata2 = 0;

if (test)
    mydata = 1;

else
    mydata = 2;

mydata2 = 0;
```

mydata contains information about test so it can no longer be Low, but mydata2 is outside the conditional, so it is untainted by test

Causes type error at compile-time
Preventing These Vulnerabilities

• What can the programmer do to secure their program?
  ‣ Decentralized Information Flow Control

• Programmer gives hints to operating system about information flows expected
  ‣ Operating system enforces those expectations

• To prevent link traversal attacks, programmers could just state that they require objects with an expected label
  ‣ And use that object safely with security types in program
Preventing These Vulnerabilities

• What can the programmer do to secure their program?
  ‣ Decentralized Information Flow Control

• Programmer gives hints to operating system about information flows expected
  ‣ Operating system enforces those expectations

• To prevent link traversal attacks, programmers could just state that they require objects with an expected label
  ‣ And use that object safely with security types in program

Example 1
```c
int{high} h1, h2;
int{low} l;
l = 5;
h2 = open({high});
h1 = read_int(h2);
```
Can All Problems Be Solved... 

• By type-safety with information flow control?

• Two limitations

  ‣ Need to **encode the security problem as an information flow problem**

  ‣ Need to **get the programmers to use type-safe languages and information flow control**

```plaintext
Insecure Program
gzip() {
    ...
    compr();
    ...
}
compr(...) {... }

Secure Program
gzip() {
    ...
    fork_compr();
    ...
}
compr(...) {
    drop();
    ...
}

gzip should always execute comp() with low cap, but always open files in main with high cap
```
Retrofitting for Security

• Take the code written in a language of the programmers’ choice (for functionality) and retrofit with security code (mostly-automated)

• Consider *authorization bypass vulnerabilities*
  
  ‣ In these vulnerabilities, programmers forget to add code to control access to program resources

What is authorization?

![Diagram showing the authorization process]

- **Resource user**
  - Operation request
  - Response

- **Resource manager**

- **Authorization Hooks**

- **Reference monitor**
  - Allowed?
  - YES/NO

- **Authorization policy**

  ⟨Alice, /etc/passwd, *File_Read*⟩
Retrofitting for Security

- To prevent authorization bypass, **all paths** from the untrusted input to the security-sensitive program operations must be mediated by an authorization hook
  
  ‣ **Complete mediation**

- Harris claim: From an insecure program and policy, we can automatically write a secure program by solving a two-player safety game.

```plaintext
Insecure Program
gzip() {
  ...
  compr();
  ...
}
compr(...) { ... }

Disallowed Executions
. [ compr() with high cap ]
. [ open() with low cap ]

Secure Program
gzip() {
  ...
  fork_compr();
  ...
}
compr(...) { drop();
  ...
}
```
Retrofitting as a Game

• Two steps:
  ‣ Model uninstrumented program, policy, and security goal (Capsicum) as languages/automata
  ‣ From automata, translate retrofitting problem to a two-player safety game

• Modeling
  ‣ **Program** is a language over program instructions (Instrs)
  ‣ **Policy** is a language of instructions paired with capabilities (Instrs × Caps)
  ‣ Capsicum is a *transducer* from instructions and primitives to capabilities (Instrs U Prims → Caps)
Retrofitting as a Game

- Two players: **Attacker** and **Defender**
- Play: **Attacker** and **Defender** choose actions in alternation
- Player goals:
  - **Attacker**: generate a play “outside policy” accepted by the game
  - **Defender**: thwart the **Attacker**
- Winning strategy: choices that a player (defender) can make to **always** win a game
Take Away

- Programming for security is difficult
  - Programmers create “flaws” that are often accessible and exploitable by adversaries (vulnerabilities)
- Program analysis can find some flaws
  - Static and dynamic, but limitations for each
- But they may not really be vulnerabilities
- The future of secure programming may look very different
  - **Now**: use favorite language for achieving function and try to add security code without creating flaws
  - **Future**: use favorite language for achieving function and retrofit based on a security program (“policy” in Capweave)