CSE 543 - Computer Security

Lecture 17 - Language-based security
October 25, 2007
URL: http://www.cse.psu.edu/~tjaeger/cse543-f07/
Engineering Disaster?

- Millions of Bots
  - Compromised applications
- Programming errors
  - Enable code insertion
- What can we do to fix them?
- Just starting to get serious...
Buffer Overflows

- One means by which the bad guys take over a host
  - install root kits
  - use as SPAM bots
  - use as zombies
  - launch other attacks
- There are many attacks, but this is most prevalent
- It all starts with some programmer mistake
  - e.g., bad software
Buffer Overflows (continued.)

- **Stack frame**
  - how local variables and program state is stored in most modern programming languages

- **The Attack**
  - overwrite buffer on stack with new return pointer pointing to adversary code
  - Return from function lands program counter into bad guys downloaded code

- **Game over -- they now control that host**
Buffer Overflow Prevention

- **StackGuard**
  - Push a ‘canary’ on the stack between the local vars and the return pointer
  - Overwrite of canary indicates a buffer overflow
  - Requires changes to the compiler

- **Q:** Would this solve the problem?

- **Thorough summary:**
Other Input Problems

• Function Pointers
  • Overwrite a local function pointer variable
  • Q: What can be done?

• Heap overflow
  • Overflow a buffer on the heap

• Integer Overflow
  • For signed 8-bit integers
    • $127+1 = ??$

• Malformed Character Input
  • What does URL “<ipaddr>/scripts/..%c0%af../winnt/system32” decode to?
Java World

- Type Safe Language
  - No buffer/heap/ptr overflows
  - No unsafe casts
  - Still have integer overflows?
- Java Virtual Machine
  - Interpret bytecodes (or compile together)
  - Security Manager (reference monitor for JVM)
- Q: What is the trust model of a Java application?
Ccured

• From C to Memory-safe C Translator
  – Find the minimum number of runtime checks to ensure memory safety

• Classify Pointers
  – Safe
  – Wild
    • Need runtime checks for wild pointers

• Runtime Checks
  – Similar to declassifiers in DLM
  – Written by hand, in general
C Analysis

• Assume Type Safety in Analysis
  – On what basis?
  – Trust that the programmer does not subvert

• Is this a reasonable assumption?
  – Unsound analysis
    • False negatives are possible
  – Sound analysis
    • If no unsafe behavior relative to analysis can be assumed

• Actually, lots of work in this area
• Used in production code: Microsoft
Source Code Analysis

• Shallow tools for bug finding
  – Prefix, Prefast -- Microsoft

• Companies that will check your code
  – Coverity -- based on MC

• Deep tools for verifying correctness
  – SLAM -- for device drivers

• Add security to legacy code
  – Generate LSM
  – Generate reference monitor for X Server

• Lots of other topics
  – Privilege separation
  – Domain transition
  – Error reporting
Enforcing security policy

- DAC
- MAC
- certificates
- trust management
- SELinux
- anti-virus
- IDS
- firewalls
- encryption
- legal measures

None of these provide end-to-end confidentiality
Information-flow control

- What is it?
- Simple security & ★-property
- Why?
- Leandro Aragoncillo, e.g.
- Problem: Information release
- Solution: Information Flow Control
- Stronger enforcement than reference monitors
Label and monitor

- **Key:**
  - tag data
  - monitor flows

- **RMs tag actual data**
  - all data/processes have label
  - central security monitor checks operations, data access against policy

- **Security-typed languages use virtual tags**
  - data types are labeled
  - type checker validates flows
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

**Example 1**

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

**Example 2**

```c
String{low} proc(Object{high} o);
...
main()
{
    Object{high} obj;
    String{low} s;
    s = proc_obj(obj);
    ...
}
```

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

Static (virtual) tagging

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

• Noninterference is too restrictive

• Examples:
  • Encryption
  • Distributed auction
  • Password check

• Solutions:
  • DLM and selective declassification
  • Robust declassification
  • Quantitative security
Open challenges

• System-wide security
• Certifying compilation
• Abstraction-violating attacks
• Dynamic policies
• Practical issues
• Variations of static analysis
“The inability to express or enforce end-to-end security policies is a serious problem with our current computing infrastructure, and language-based techniques appear to be essential to any solution to this problem.”