Program Language Security

CSE443 - Spring 2012
Introduction to Computer and Network Security
Professor Jaeger

www.cse.psu.edu/~tjaeger/cse443-s12/
Prevent Return-Oriented Attacks

• What is the cause of these attacks?
Prevent Return-Oriented Attacks

• What is the cause of these attacks?

• Adversary uses a vulnerability to gain control of the program
  – In particular the program execution stack
  – By using the stack, the adversary can control the sequence of instructions executed
    • Can run any instructions they want

• So, how would we prevent these attacks?
Prevent Return-Oriented Attacks

• What is the cause of these attacks?

• Adversary uses a vulnerability to gain control of the program
  – In particular the program execution stack
  – By using the stack, the adversary can control the sequence of instructions executed
    • Can run any instructions they want

• So, how would we prevent these attacks?
  – Prevent the adversary from changing the control flow of the program from its legitimate paths
  – Ensure that all program operations satisfy a security policy
Control Flow Integrity

• **Goal**: Ensure that all stack control operations are legal

• What are “stack control operations”?
  – Calls
  – Returns

• Calls
  – Only call instructions that are possible given the source code
    • Normal function calls
    • Function pointers

• Return
  – All functions return to expected return addresses
    • May be more than one though

• If CFI is enforced then the adversary can only choose instruction sequences authorized by the program
Software Control Flow Integrity
Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with:
Úlfar Erlingsson and Martín Abadi
Control Flow Integrity

Attack Model

**Powerful Attacker:** Can at any time arbitrarily overwrite any data memory and (most) registers
- Attacker cannot directly modify the PC
- Attacker cannot modify our reserved registers (in the handful of places where we need them)

**Few Assumptions:**
- Data memory is Non-Executable *
- Code memory is Non-Writable *
- Also… currently limited to whole-program guarantees (still figuring out how to do dynamic loading of DLLs)
Control Flow Integrity

Our Mechanism

\[
\text{FA} \quad \text{FB}
\]

\[
\begin{align*}
\text{call fp} & \quad \text{nop IMM}_2 \\
\text{if(*fp \neq \text{nop IMM}_1) halt} & \\
\text{nop IMM}_1 & \quad \text{if(**esp \neq \text{nop IMM}_2) halt} \\
\text{return} & \\
\end{align*}
\]

NB: Need to ensure bit patterns for nops appear nowhere else in code memory

CFG excerpt

\[
A_{\text{call}} \rightarrow B_1
\]

\[
A_{\text{call+1}} \leftarrow B_{\text{ret}}
\]
More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int→int function

$$\text{CFG excerpt}$$

$A_{\text{call}} \rightarrow B_1 \rightarrow C_1$

$succ(A_{\text{call}}) = \{B_1, C_1\}$

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction
Imprecise Return Information

Q: What if $F_B$ can return to many functions?
A: Imprecise CFG

CFG excerpt

```plaintext
A_{call+1} \leftarrow B_{ret}
D_{call+1} \leftarrow B_{ret}
\text{succ}(B_{ret}) = \{A_{call+1}, D_{call+1}\}
```

CFG Integrity:
Changes to the PC are only to valid successor PCs, per succ().
Control Flow Integrity

No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

A\text{call} \rightarrow B_1
E\text{call} \rightarrow C_1

CFG excerpt

A\text{call} \rightarrow B_1
E\text{call} \rightarrow C_{1E}
C_{1A}
Secrecy Properties

• **Simple-Security Property (Read-Down)**
  – A subject $s$ can only read from an object $o$ if the subject’s clearance *dominates or is same as* the access class of the object

• ***-Security Property (Write-Up)**
  – A subject $s$ can only write to an object $o$ if the subject’s clearance *is dominated by or is the same as* the access class of the object
**Program-Level Secrecy**

- **Situation**: A program reads data at a higher access class than it writes
  - Program can leak secret that it reads by writing it to the lower access class

- **Challenge**: Write a program where you can prove that no illegal *information flows* (i.e., violating MLS properties) can occur
Denning Lattice Model

- Information flow within a program
  - Can a secret variable leak to a public variable?
- Model covers all programs
  - Statement S
  - Sequence S1, S2
  - Conditional c: S1, ..., Sm
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)

• Implicit flows only occur in conditionals
Denning 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
Information Flow

- Explicit and implicit flows form a graph

Stmt
b = a

Cond
if (b < 1)

Stmt
c = f

Stmt
c = d

Stmt
e = c

Stmt

• Resulting flow graph

a

b

c

d

f

e
Secure When...

- Suppose $e$ is public
  - What variables can be secret?
- Suppose $d$ is public
  - What variables can be secret?
- Suppose $b$ is secret
  - What variables can be public?
Making Program Information Flow Practical

- Languages have been extended with these semantics
  - Java
  - OCaml
- Annotate data types with security labels
  - Int becomes “Secret Int”
  - Where “Secret” is a label in a secrecy lattice
- Compilation is successful only when the program “type checks”
  - That is, all program expressions must satisfy typing rules
  - Where typing rules capture explicit and implicit flows
- Any limits to which languages can be supported?
Making Program Information Flow Practical

• Need a strongly typed language
  – C is not supported
  – Although there are other tools to verify type safety

• In general, the problem is that there are too many information flow errors in programs
  – Especially implicit flows

• People have found it difficult to take existing programs and make them safe for information flow
  – And programming from scratch is not easy either
  – Still for experts...
Take Away

• To prevent return-oriented programming
  – Enforce control flow integrity
  – Enforce stack control operations relative to program semantics
    • Some imprecision must be dealt with

• However, we may also need to obtain secrecy guarantees for more privileged programs
  – Ones that could violate MLS
  – Use Denning’s model to prove compliance
    • Information flows: Explicit and implicit