Discovering Application-level Insider Attacks using Symbolic Execution

Karthik Pattabiraman
Zbigniew Kalbarczyk
Ravishankar K. Iyer

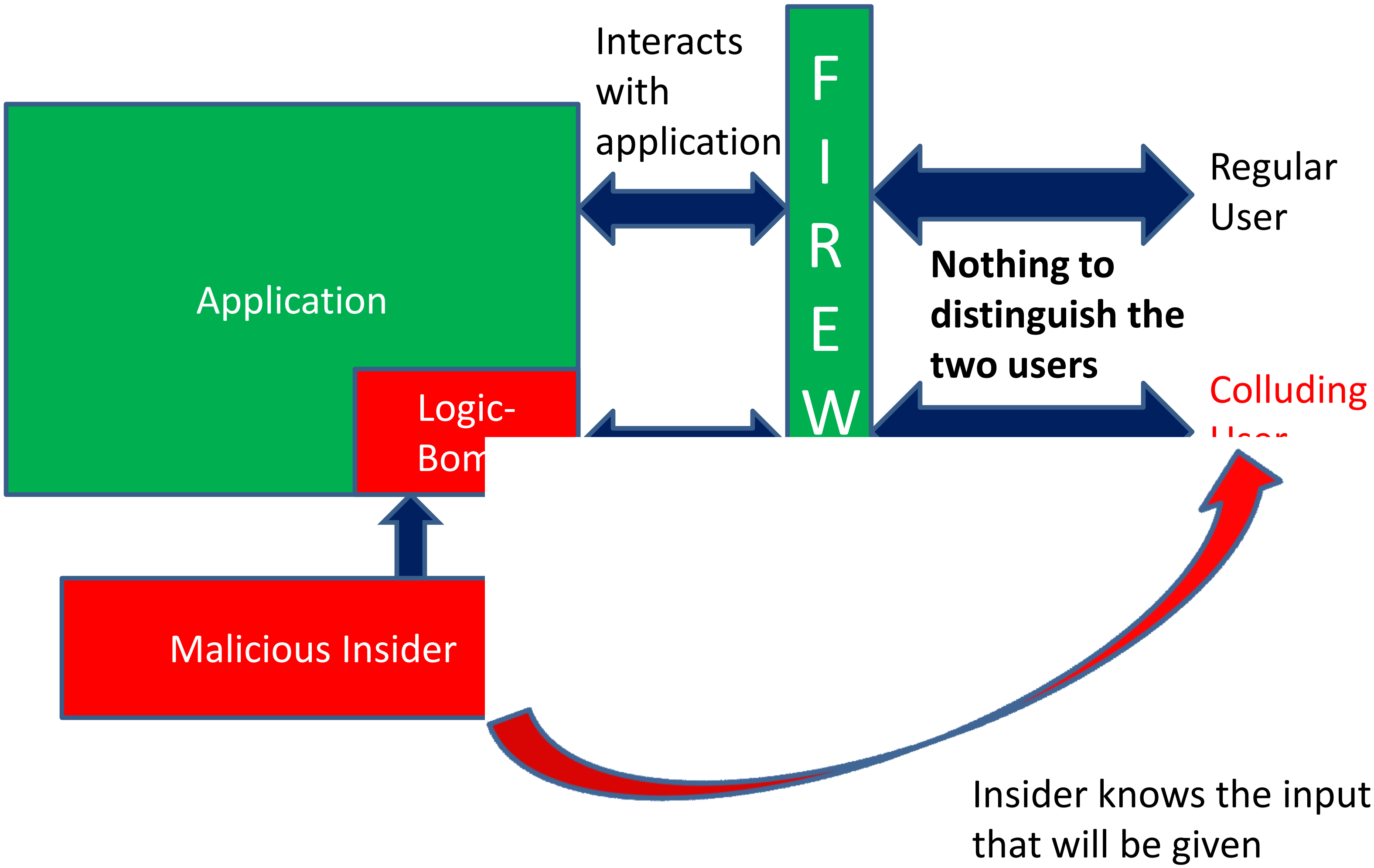
Motivation: Insider Attacks

- **Malicious insiders can plant logic-bombs/back-doors in apps**
  - Many libraries distributed in binary form (source unavailable)
  - Even if source is available, original developer may have left and nobody understands the code anymore
  - Outsourcing/off-shoring compound the problem
- **Both closed-source and open-source equally vulnerable**
  - Study of 100 closed-source packages found 79 had dead-code and 23 had unwanted code (back-doors) [Veracode '09]
  - Open-source no panacea (attempts to plant backdoor in the Linux kernel took 4 days to discover – may be more for less freq. used S/W)
- **Malicious system-administrators can modify/recompile code**
  - Widely-penetrated fraud scheme in organization went undetected
  - Sys. Admin commented out a single line of source code [CERT'09]

Application-level Insiders

- **Insider can corrupt both registers and memory**
  - Malicious third-party library or plugin
  - Logic flaw planted by disgruntled programmer
  - Malicious operating system/higher privileged process
- **Insider wants to elude detection (as far as possible)**
  - Cannot directly execute code that hijacks application
  - Cannot perform large-scale corruptions of app data
- **Insider does not want to crash the application**
  - Denial-of-service attacks not considered

Attack Scenario



Existing Techniques

Symbolic Execution: Generating attack inputs for known vulnerabilities [EXE’06][Bouncer’07]

Attack Graphs: Model attackers at the network-node level [Jha’01][Upadhyaya’04]

Need for a formal framework to automatically explore all possible insider attacks on the application at the code-level

Static Analysis: Finding vulnerabilities in programs [SPLINT’01][MOPS’04]

Process Calculii: Model attackers at process level [Probst’06]

Problem Statement

- **Given a program and a set of attack points, can we discover all possible insider attacks to achieve a certain goal (for the attacker) ?**
  - E.g. Make the program print "authenticated" even if wrong password is supplied by the user
  - Identify both the data item to be corrupted (AND) the precise value that it must be corrupted with

- **Key Idea: Symbolically execute program under all possible malicious value perturbations**

Assumptions

- **Attacker can corrupt a single data item at specific points in the program execution**
  - Data item can be register/memory address
  - Control-data can also be corrupted e.g. function ptrs
- **Only one corruption allowed per run, but corrupted value can be any valid program value**
  - Value must be represented in the assembly code
- **Corruption only allowed at fixed program points (attack points), e.g. Calls to 3rd party functions**

SymPLAID: Approach

- **Goal: Explore all insider attacks that may be launched in an application (expressed in assembly language)**

- **Attack Model**
  - Attacker may corrupt **any data** in program (stack/heap/reg.)
  - Attacker has a specific **goal state** (in terms of the application)
  - Attacker launches attack at **attack points** in applications

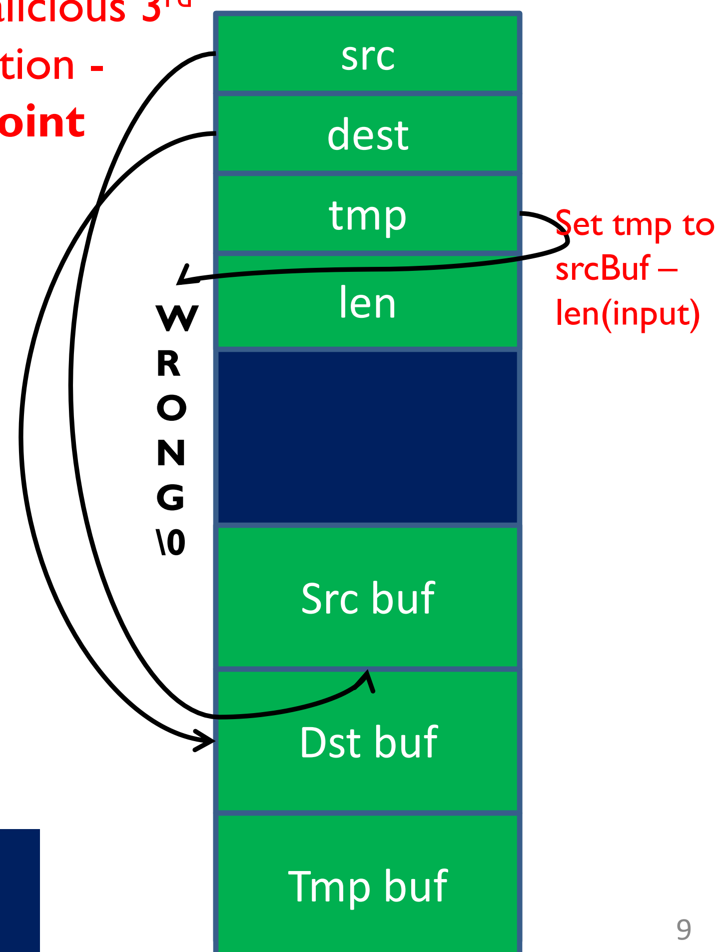
- **Output:** Enumeration of **all** possible attacks in the model that lead to the attacker's goal undetected

Insider Attack Example

```
void authenticate(void* src, void* dest, void*
   temp, int len){
  readInput1(temp);
  strncpy(src, temp, len);
  readInput2(temp);
  untrusted_function();
  strncpy(dest, temp, len);
  if (! strncmp(dest, src, len) )
       return 1;
  return 0;
}
```

Call to malicious 3rd party function - **Attack point**

Both point to empty strings



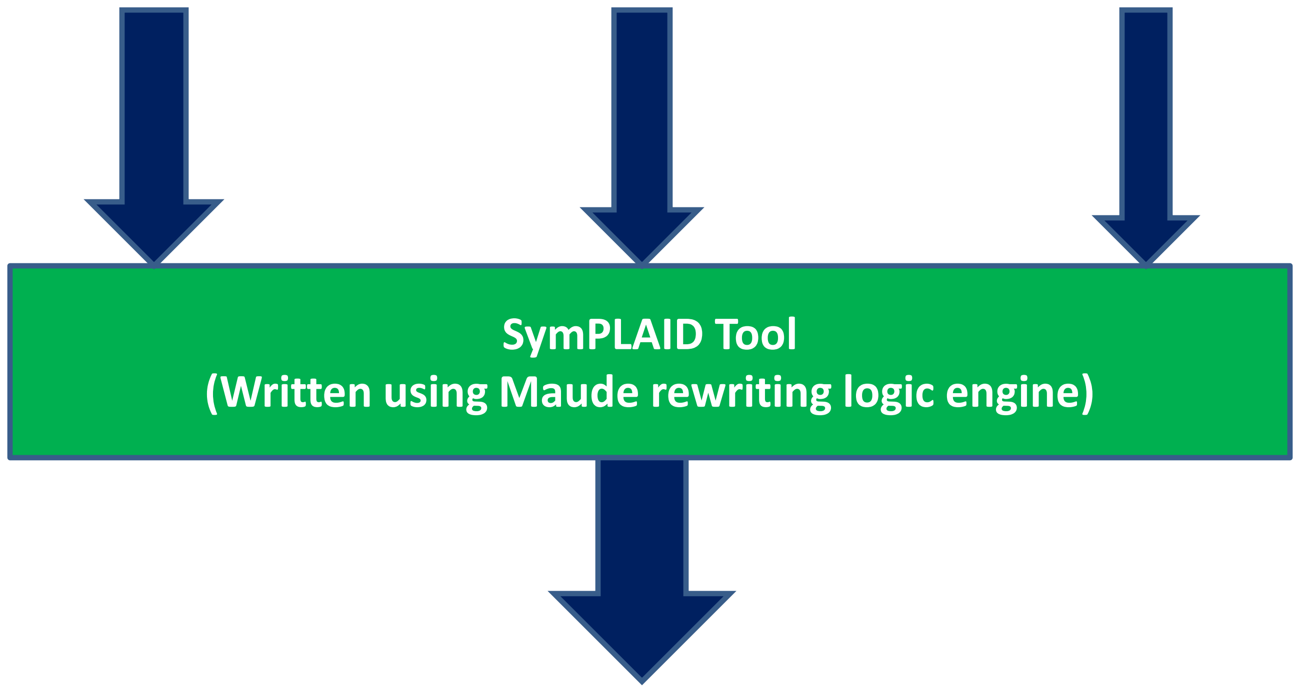
Attacker's Goal: Make the above function return 1 even if the wrong password is given

SymPLAID: Tool

Program expressed in assembly language

Set of attack points in the application

Attacker's goal as a first-order logic formula



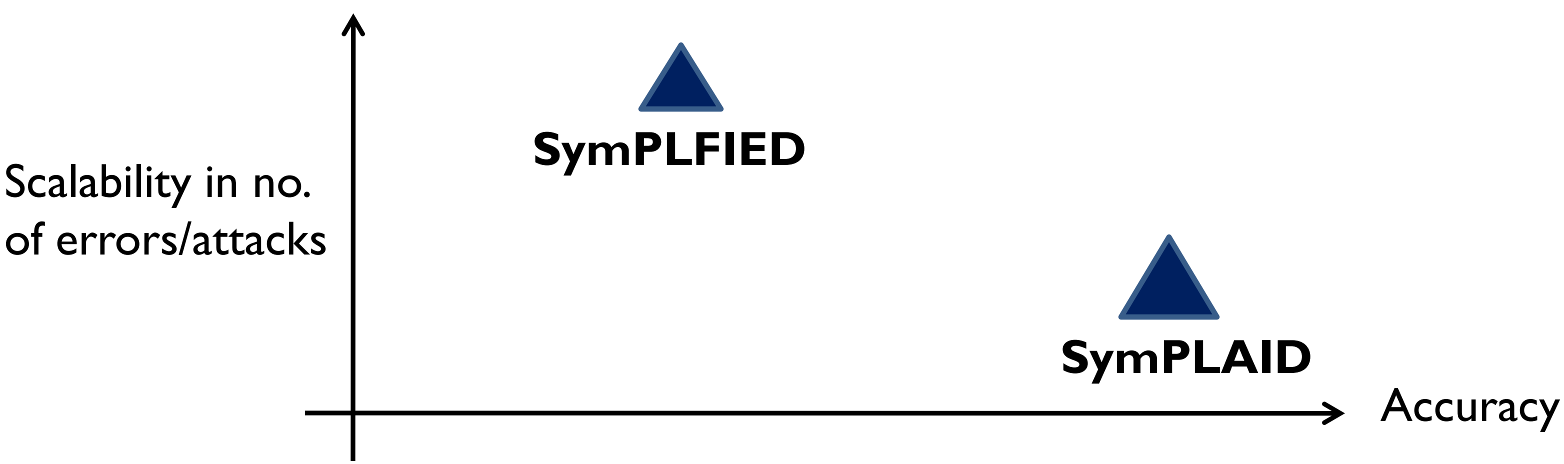
Comprehensive enumeration of insider attacks that can be launched on the program at the specified attack points and lead to the specified goal state

SymPLAID: Difference with SymPLFIED

- **SymPLFIED**
  - More concerned about effect of the error than its origins

- **SymPLAID**
  - Both the origin and effect of the security attack
  - Tracks each value

SymPLFIED emphasizes scalability over accuracy for reasoning about errors
SymPLAID emphasizes accuracy over scalability for reasoning about attacks

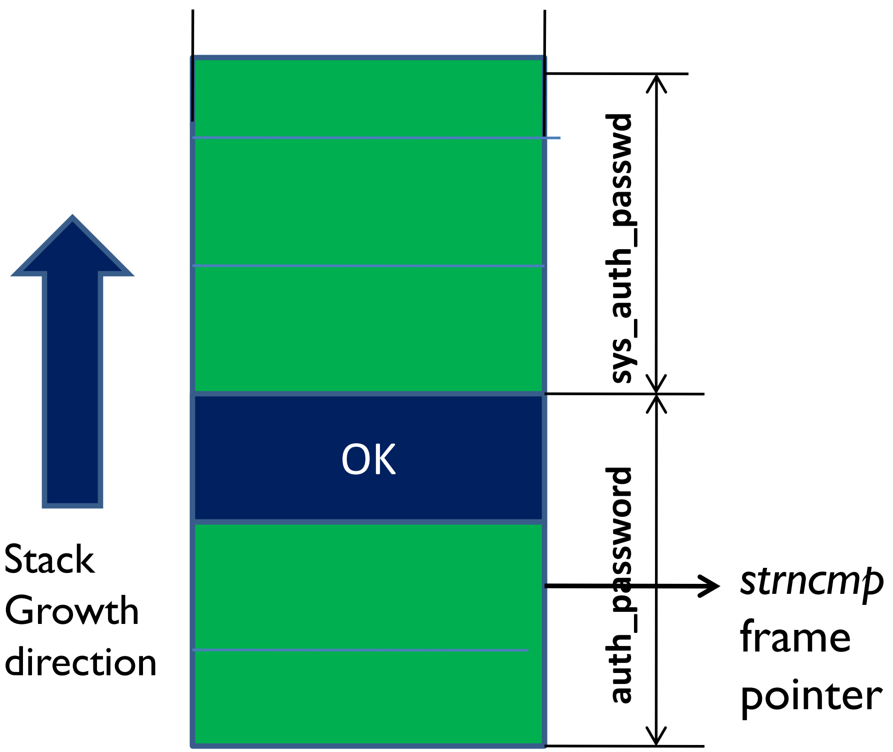


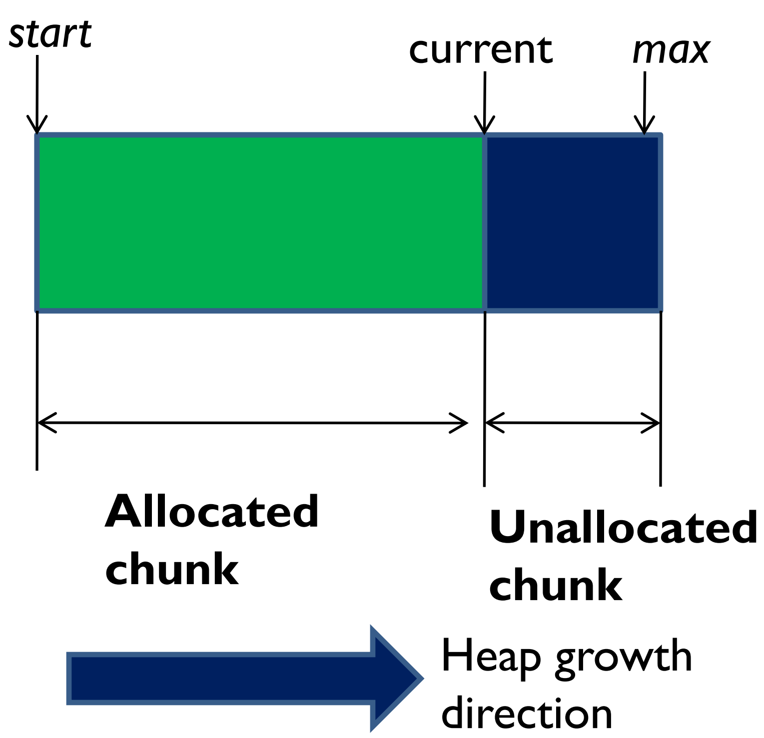
SymPLAID: Case Study

- **Demonstrated on SSH authentication stub**
  - 200 lines of C code, 500 assembly language instructions
  - Checks if user name is in list of allowed users, AND
  - Checks if user password matches system password
- **Attacker Goal: To authenticate him/herself with**
  - Wrong username, Wrong password
  - Wrong username, Correct password (= default password)
  - Correct username, Wrong password
- **Ran task on a 50 node AMD Opteron cluster**
  - Ran for approximately two full days (maximum of all times)
  - Equivalent time to running on a single node for a month

SymPLAID: Case Study Results

- **Real Attack Example**
  - Overwriting stack/frame pointers of calling functions
- **Spurious Attack Example**
  - Overwriting the *current* variable in chunk allocator





Summary

- **SymPLAID: Formal technique to systematically consider effect of security attacks on programs**
  – Generate all possible insider attacks for a given goal
  – Can guide development of defense mechanisms

- **Tracks value corruptions at assembly code level**
  – Attacker can corrupt program value(s) at specific points in the program (attack points)

- **Demonstrated on real application (OpenSSH) to find non-intuitive attack scenarios**

Future Directions

- **Scale the technique to larger programs**
  - Requires efficient constraint-solving capabilities
  - Truncate paths that do not seem "promising"
- **Eliminate the need to specify the attack goal**
  - Dictionary of common attack goals in applications
  - Specify good behavior rather than bad behavior
- **Technique to protect apps from insider attacks**
  - *Information-Flow Signatures (IFS)* to protect security critical data in applications using static analysis