Building Systems to Enforce Measurable Security Goals

Trent Jaeger
Penn State University, SIIS Lab, CSE Dept.

Zurich Information Security Center Workshop
September 5, 2008
SIIS Laboratory (http://siis.cse.psu.edu)

Faculty
Prof. Trent Jaeger (tjaeger@cse.psu.edu), Co-director
  operating systems security, policy design and analysis, source code analysis
Prof. Patrick McDaniel (mcdaniel@cse.psu.edu), Co-director
  network security, security modeling, critical infrastructure, security-typed languages, formal security policy
Prof. Sencun Zhu (szhu@cse.psu.edu)
  ad hoc and sensor networks, buffer-overflow and worm, p2p security
Prof. Adam Smith (asmith@cse.psu.edu)
  cryptography, algorithms, privacy-preserving computation

Funding:
- National Science Foundation
- Army Research Office/DOD
- DTO/IARPA
- CISCO, Raytheon (NSRC)
- Motorola (SERC), Samsung
- IBM Research

Ongoing Projects:
- Secure Storage Systems
- Language Based Security
- Systems and VM Security
- Telecommunications Security
- Self-healing Sensor Networks
- Privacy-preserving Computing

- Students (Over 20 graduate students)

Factoids: Established September 2004,
Location - 344 IST Building - contact siislab@cse.psu.edu
Intended “Take Aways”

• Information Flow Policy Models
  ‣ Too idealized for using in “pure” sense, but still useful as a goal to test against

• Application-level Policy Enforcement
  ‣ Cannot (never did) only trust the OS for access control, and application development is getting ready to meet the challenge

• Layered Enforcement
  ‣ With the addition of a VMM layer, and better app enforcement, we need an approach to connect these layers
What Are Systems?

Appl

Appl

Appl
What Are Systems?

Applications

Operating System
What Are Systems?

- Application policy enforcement: databases, JVM, X Windows, daemons, browsers, email clients, servers
What Are Systems?

- Application
  - Policy
- Operating System
- Virtual Machine Monitors
- Network
• Historically, OS treats applications as black boxes
  ‣ OS controls flows among applications
  ‣ Security requirements determined by allowed flows
Idealized Security

- **Multilevel Security (MLS) for secrecy**
  - **Secrecy requirement**: Do not leak data to unauthorized principals
  - Only permit information to flow from less secret to more secret principals/objects
  - E.g., Can only read a file if your clearance dominates that of the file

- **Biba Integrity**
  - **Integrity requirement**: Do not depend on data from lower integrity principals
  - Only permit information to flow from high integrity to lower integrity
  - E.g., Can only read a file if your integrity level is dominated by the file’s
Information Flows

- **Secrecy** (MLS): If the OS permits a secret application/object to flow to a public application/object, then there may be a leak (e.g., Trojan horse)
  
  ![Diagram](Secret→Public)

- **Integrity** (Biba): If the OS permits a low integrity input to flow to a high integrity application/object, then there may be a dependency (e.g., buffer overflow)
  
  ![Diagram](Low→High)
Practical vs. Ideal

- Do these idealized approaches based on information flow enable practical realization of OS enforcement?
- Secrecy is possible in some environments
  - Implemented in a paper world, previously
- Integrity has not been realized in practice
  - Many processes provide high integrity services to others
- Result: Depend on many applications to manage information flows
Example: logrotate

- **Logrotate** is a service that swaps logs
- It rotates logs through sequence
  - Secrecy: Logs may span all security levels on system
  - Thus, *logrotate* is trusted in SELinux
- It reads a configuration to tell it what to do
  - Integrity: Logs must not leak into configuration files
  - Configurations must not cause file leakage
**SELinux/MLS Trusted Programs**

- **The OS trusts** that privileged applications preserve system secrecy (30+ programs)

SELinux/MLS:

<table>
<thead>
<tr>
<th>Category</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy management tools</td>
<td>secadm, load_policy, setrans, setfiles, semanage, restorecon, newrole</td>
</tr>
<tr>
<td>Startup utilities</td>
<td>bootloader, initrc, init, local_login</td>
</tr>
<tr>
<td>File tools</td>
<td>dpkg_script, dpkg, rpm, mount, fsadm</td>
</tr>
<tr>
<td>Network utilities</td>
<td>iptables, sshd, remote_login, NetworkManager</td>
</tr>
<tr>
<td>Auditing, logging services</td>
<td>logrotate, klogd, auditd, auditctl</td>
</tr>
<tr>
<td>Hardware, device mgmt</td>
<td>hald, dmidecode, udev, kudzu</td>
</tr>
<tr>
<td>Miscellaneous services</td>
<td>passwd, tmpreaper, insmod, getty, consoletype, pam_console</td>
</tr>
</tbody>
</table>
Situation Is Much Worse

- **Clients**
  - Lots of client programs are entrusted with information with different secrecy/integrity requirements
  - Email, browser, IM, VOIP, …

- **Servers**
  - Historically, many servers have enforced security policies because they handle multiple clients
  - Web servers, databases, mail, repositories, …

- *Information flow alone is not enough to build a secure system!*
What Do We Do?

- **Unsatisfying Solution #1: Ignore Exceptions**
  - Processes are outside the model
  - E.g., so-called “Trusted Readers/Writers” in MLS
  - **Result:** Accept that some processes can violate policy, often blindly

- **Unsatisfying Solution #2: Dump Info Flow Entirely**
  - Change to a more general policy model
  - No meaningful security goal (is least privilege adequate?)
  - **Result:** Systems are complex -- 50K SELinux rules
Reference Monitor Concept

• System must fulfill reference monitor concept:
  ▶ Complete mediation
    • All security-sensitive operations must be mediated by the authorization policy
  ▶ Tamperproof guarantees
    • The trusted computing base of the system that authorizes access must be protected from tampering by untrusted processes
  ▶ Verifiability
    • The reference monitor must be simple enough to be verified
Reference Monitor Concept

• Meeting reference monitor concept guarantees is our challenge:

  ‣ **Complete mediation**
  
  • All security-sensitive operations must be mediated by the authorization policy
  • Security-sensitive operations are in VMM, OS, application (Layering)

  ‣ **Tamperproof guarantees**
  
  • The trusted computing base of the system that authorizes access must be protected from tampering by untrusted processes
  • All trusted code and policy must be protected from unauthorized modification (TCB Definition and Integrity)

  ‣ **Verifiability**
  
  • The reference monitor must be simple enough to be verified
  • The system’s enforcement must be verifiable against meaningful security goals (Goal Definition and Compliance)
Goals

- Determine an approach to building systems (by layering applications, OS, VMM) such that they satisfy the reference monitor guarantees
  - Define a TCB perimeter in practical systems
  - Enable applications to aid in the protection of that perimeter
  - Ensure that applications enforce system security goals
Experiments

- We have investigated a few candidate approaches
  - Trying to move closer each time

- Experiment #1 (Analyzing integrity protection in the SELinux example policy, USENIX Security Symp, 2003)
  - Assume information flow goal is ideal; Design policy as close to ideal as practical and trust applications

- Experiment #2 (Toward automated information-flow integrity verification in security-critical applications, NDSS 2006)
  - Propose modification to information flow ideal; Enable system and applications to support this approach

- Experiment #3 (Verifying compliance of trusted programs, USENIX Security Symp, 2008)
  - Applications and system enforce information flow ideal; Enable deployment of application on system automatically

- Future?
Experiment #1: Find SELinux TCB

• Can we identify a TCB in SELinux Example Policy whose integrity protection can be managed (circa Linux 2.4.19)?
  ‣ See [USENIX Security 2003]

• Tasks:
  ‣ Can We Identify Trusted Programs?
  ‣ Can We Define a Security Goal to Protect These Programs?
  ‣ Can We Verify This Goal?
  ‣ How Do We Debug Conflicts?
Proposed Approach

- Propose a TCB from SELinux subjects
- Identify Biba integrity violations
- “Handle” integrity violations
  - Classify integrity violations
  - Remove violations that can be managed by application
    - Application is trusted to protect itself
  - Revise TCB proposal
    - Revise SELinux policy
- Result: All information flows are legal or accounted
Propose a TCB

- SELinux Policy (way oversimplified)
  - Type Enforcement [Boebert and Kain 1985]
  - Mandatory Access Control Model
  - Subjects and objects have labels (types) and policy defines access of subjects to objects

- Trusted Computing Base
  - Processes that we must trust to enforce system policy
  - These processes correspond to SELinux subject types
Propose a TCB - Detail

• Use SELinux transition graph (i.e., who can exec whom) for server programs (e.g., httpd_t) to identify base subject types

• Ones that provide TCB services (e.g., authentication)

• Others that have many transitions (hard to contain)
Identify Integrity Violations

• Biba Integrity Analysis -- Gokyo, PAL, PALMS

• TCB subject types $\rightarrow$ read/exec perms
  ‣ Generate corresponding “integrity-sensitive write” perms

• Others $\rightarrow$ write perms
  ‣ Generate corresponding “integrity-sensitive read” perms

• Analysis
  ‣ Do Others’ write to integrity-sensitive writes?
  ‣ Do TCB Subjects read from integrity-sensitive reads?
Integrity Analysis

High Subject

Subject

Perm

Object Read

Low Subject

Subject

Perm

Object Write

Low Subject Can Modify Input To High
Are There Integrity Violations?

- For Linux 2.4.19 -- SELinux Strict Policy
- Permissions
  - 129 perms used to “read down”
    - 57 socket perms, 25 fifo perms
  - 1583 perms used to “write up”
- Subjects
  - 28 high integrity subjects “read down”
    - 35 for sysadm_t, 4 for load_policy_t
  - 150 low integrity subjects “write up”
The subject-permission assignments that lead to a conflict result in a **minimal cover** of all conflicts.
Example Resolutions

```
High Subject Type

Attr Perm

Perm

Exclude Object Type
Deny Access

Exclude Subject Type

No Deny Read

Attr

Perm

S-P Assign

Conflicts

S-P Assign
```
Integrity Resolutions

• Remove Subject Type or Object Type
• Reclassify Subject Type of Object Type
• Change Subject Type-Permission assignment

• Clark-Wilson reads
  ‣ Allow reading of low integrity data that meet Clark-Wilson

• Deny Object Access
  ‣ Track low integrity writes per object

• LOMAC Subject Type (sysadm)
  ‣ Reduce integrity level of subject when reading low integrity data
• Conclusions
  ‣ Biba Information Flow Integrity
    • May not be so far off practical
    • But, we cannot force Biba (or other ideal models, e.g., LOMAC)
  ‣ Need to address conflicts
    • Identify resolutions

• Issues
  ‣ Comprehensive Analysis is Possible, but Low-Level (Who is going to do it?)
  ‣ Resolution Selection Is Ad Hoc (Depends on applications)
Experiment #2: Deploy High Integrity App [NDSS 2006]

Integrity property: Trusted processes don’t depend on untrusted ones

- Untrusted user process
- setuid-root cron job
- sshd_config

File permissions don’t reveal the problem

[Provos et al USENIX 2003]
Legal vs. Illegal Flows

Existing models either:

a) don’t correctly classify
b) require extra work

User process

Privileged OpenSSH

Filter

Root shell

Bad case:

Unprivileged OpenSSH

Network
Our New Integrity Model: CW-Lite

- Preserve info-flow intention of Clark-Wilson
  - Filter untrusted inputs to trusted processes
    - For all interfaces, trusted processes must reject untrusted inputs or upgrade their integrity immediately
    - Trusted processes are certified to perform correct operations
- But relax these two constraints:
  - Don’t require all interfaces to perform filtering
  - Check existence of filters, not correctness
    - Ultimately, check correctness of filters manually, not entire program
Add Filtering Interfaces

Root shell → Privileged OpenSSH → Unprivileged OpenSSH → Network

Filter is invisible to OS

Bad case:

User shell
Exposing Filtering Interfaces

• MAC system can’t see filtering interfaces
  ‣ Permissions are per-process, not per-interface

• Solution: Send hint from inside the process
  ‣ Programmer adds annotation to filtered interface

• Use two subject types for each process
  ‣ *Default subject type* allows inputs only from TCB
  ‣ Filtering interfaces use *filtering subject type* which enables additional permissions
For OpenSSH

- Root shell
- Privileged OpenSSH
  - Privileged OpenSSH (filtering)
  - User shell
- Unprivileged OpenSSH

Bad case:

Network
Enabling Filtering Subject Types

- **SELinux kernel mod** enables two subject types (default & filtering) for each process
- **User library extension** adds
  - Ability to switch between both subject types
  - **DO_FILTER** convenience macro

  
  - **DO_FILTER**\(_{(f() \text{ := } \text{Enable filtering subject type})}
  \text{Call } f()\)
  \text{Disable filtering subject type}
Filtering Interface Example

**BEFORE**

- **Source Code**
  
  - `conn = accept()`
  
  - `// accept() fails`
  
  - `get_request_sanitized(conn)`

- **Security Policy (default DENY)**
  
  - Apache: ALLOW read httpd.conf
  
  - `// Problem: network not in TCB`
  
  - Apache: ALLOW accept

**AFTER**

- **Source Code**
  
  - `DO_FILTER(conn = accept())`
  
  - `// accept() succeeds`
  
  - `get_request_sanitized(conn)`

- **Security Policy (default DENY)**
  
  - Apache: ALLOW read httpd.conf
  
  - `// Apache-filter: non-TCB OK`
  
  - **Apache-filter**: ALLOW accept
Who Has To Do What?

**Developer**

- Identify filtering interfaces
- Add `DO_FILTER` annotation
- Split permissions among two subject types

**System Administrator**

- Choose a TCB one time for all apps
- Run Gokyo on security policy

**Errors?**

- **No** → Done
- **Yes** → Fix Errors:
  1. Remove offending apps
  2. Remove perms
  3. Add to TCB
Experiment #2 Summary

• Conclusions
  ‣ Provide an approach for application developers to justify their program’s use of untrusted (low integrity) inputs
  ‣ Integrate this justification with the approach for verifying system integrity

• Issues
  ‣ Need to modify SELinux system to recognize filtering subjects
  ‣ Don’t know that the application OpenSSH is enforcing the system’s security goals
We want to download and install applications and ensure that they can enforce system security goals [USENIX Security 2008]
Trusted Programs

- They are expected to only perform safe operations even though they have the rights to perform unsafe operations.

- Examples:
  - Startup utilities, File management tools, Network utilities, Auditing and logging services and Hardware, Device, User and Policy Management Software.

- Current SELinux systems include 30+ trusted applications.
Example: logrotate

- **Logrotate** is a service that swaps logs
- It rotates logs through sequence
  - Secrecy: Logs may span all security levels on system
  - Thus, logrotate is trusted in SELinux
- It reads a configuration to tell it what to do
  - Integrity: Logs must not leak into configuration files
  - Configurations must not cause file leakage
Trust in Trusted Programs

• Currently, we blindly trust these applications.

• Now, we do not have to:
  ‣ Security-typed languages
  ‣ User-level reference monitors

• We want to ensure that the trusted program complies with the system security policy
  ‣ Mandatory access controls implemented at Operating System, Virtual Machine Monitor and Network layers (SELinux, Trusted Solaris, SEDarwin)
Information Flow Control

- Trusted programs have their own policies; We can translate those policies into information flow graphs.

- Nodes represent subjects and objects in a given policy
- Edges represent information flows allowed by the policy
- Each node has an associated label. It indicates security requirements for that node.
Compliance

• An information flow policy A is compliant with an information flow policy B, if:

\[ \text{Flows}_A \subseteq \text{Flows}_B \]

\[ \text{Flows}_{\text{trustedProgram}} \subseteq \text{Flows}_{\text{system}} \] \[ \rightarrow \] Trusted Program enforces System Policy

• For a trusted program:

\[ \text{G}_A \text{ and } \text{G}_B : \text{compliant} \]
\[ \text{G}_A' \text{ and } \text{G}_B : \text{non compliant} \]
Integrate System and App Policies

• We need to establish a relationship between elements of the policies [HRJM07]
  ‣ Example: (logrotate)
    • [.*:*:logrotate_var_lib_t:s0] -> configP;
    • configP -> logP;
    • logP -> [.*:*:var_log_t::*]
  ‣ Hand-designed policy
    • Manual specification does not scale
Automation of Approach

1. **Build Program** (include policy)
2. **Compose Policies** (Program + System)
3. **Verify Compliance** (Against Ref Monitor)
4. **Install / Run Program**

Automatic procedures
Composing Policy

• Compose Trusted Program Policy for system

  Jif TP

  \[\text{Prog-Policy} \rightarrow + \rightarrow \text{Sys-Policy}\]

  \[\text{Program Policy that satisfies system security goals}\]

  manual

• Compose the System Policy including program

  SELinux Module

  \[\text{ProgSys-Policy} \rightarrow + \rightarrow \text{Sys-Policy}\]

  \[\text{System Policy that tamper protects program}\]
Program Integrity Dominates

- Integrity of program components dominates integrity of data (PIDSI)
  - Map program labels to **HIGH** by default
  - Data labels are defined in **System Info Flow Graph**
  - Only allow flows from program data to system information flow graph. Some well-defined exceptions (init, installers).
Policy Composition for \textit{Logrotate}

\begin{itemize}
  \item \textbf{H}
    \begin{itemize}
      \item Config Files
      \item Executables
    \end{itemize}
  \item \textit{(S)}
    \begin{itemize}
      \item Log Files
    \end{itemize}
  \item \textbf{L}
\end{itemize}

\item \textbf{(P)}
  \begin{itemize}
    \item Executables
    \item Config Files
  \end{itemize}
Implications

• Key result:
  ‣ Can integrate trusted program policy with any MAC system
  ‣ And it will comply with system policy

• For a Jif trusted program on SELinux
  ‣ Trusted program uses channel abstraction to obtain SELinux labels [HMM07]
  ‣ Trusted program uses Jif lattice to authorize [HRJM07]
    • Can determine relationship between program labels and system labels
    • But uses system policy when only system labels are involved
Experiment #3 Summary

• Conclusions
  ‣ Trusted programs have a natural integrity relationship with the data they process
  ‣ We can use this to automate the integration of program and system policies
  ‣ We can use policy analysis to verify that trusted programs cannot be tampered on deployment [Not discussed]

• Issues
  ‣ Use CW-Lite to address use of low integrity data
  ‣ Still need to resolve a few conflicts [Not discussed]
Put It All Together

• Information Flow Policy Models
  ‣ Provide an ideal security goal (Define Trust Perimeter -- Manual)
  ‣ Determine how your system differs from that goal and resolve the security of that difference (Policy analysis -- Gokyo, PALMS)

• Application-level Policy Enforcement
  ‣ If an application must enforce system security goals (i.e., is a trusted program) integrate that enforcement with the system (PIDS1)
  ‣ If an application must ensure ideal security by protecting itself from low integrity inputs, define use of filtering interface (CW-Lite)

• Layered Enforcement
  ‣ In the construction of VM systems that enforce system security goals enable the construction of compliant VMs,OS, applications (future [CSAW08])
The major insight is that information flow enforcement, which was historically focused in operating systems, can be unified across virtual machine, OS, and application layers to achieve more flexible and effective enforcement than the OS alone.
Security at Each Layer

Multi-layer Systems

How do we ensure that all layers comply?
VM System

- Create new Browser VM
- Launch compliant FlowwolF browser
- Process Web Request

`Run Web App` → **VMM -- With Services to Construct Compliant VMs** → `Network`

[Diagram showing two SELinux VMs with FlowwolF Browsers and app info flows]
VM System

- Dom0 has services for loading a new VM for an application
Summary

- **Goal:** *Build applications and configure systems, including MAC policies, to ensure enforcement of system security goals*

- **Information Flow**
  - *It’s ideal, but it is a useful basis for security goals (Analysis Tools)*

- **Application-level Enforcement**
  - *Many applications are trusted (for system integrity and enforcing system goals), so this trust should be articulated and verified (CW-Lite and Compliance)*

- **Integrate Enforcement**
  - *Multiple layers are responsible for enforcing system goals, so need to integrate automatically (PIDS1 and VM configuration)*
Questions

• Penn State SIIS Lab
  ‣ http://siis.cse.psu.edu/

• Email
  ‣ tjaeger@cse.psu.edu