Advanced Systems Security
Retrofitting for Security

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Retroactive Security

- “Penetrate and patch” as flaws are exposed as vulnerabilities
Retroactive Security

• Several codebases have been extended with security features retroactively
  ‣ X Server, postgres, Apache, OpenSSH, Linux Kernel, browsers, etc.

• With a variety of security controls:
  ‣ Privilege separation, Authentication, Auditing, Authorization, etc.
Authorizing Access

Resource user

Operation request  ∫  Response

Resource manager

Reference monitor

Allowed?  ∫  YES/NO

⟨Alice, /etc/passwd, File_Read⟩
Authorizing Access

Resource user

Operation request ➔ Response

Resource manager

Authorization Hooks

Reference monitor

Allowed? ➔ YES/NO

Authorization policy
Retrofitting is Hard

• For authorization
  ‣ X11 ~ proposed 2003, upstreamed 2007, changing to date. [Kilpatrick et al., ‘03]
  ‣ Linux Security Modules ~ 2 years [Wright et al., ‘02]

Painstaking, manual procedure

At this point, SE-PostgreSQL has taken up a *lot* of community resources, not to mention an enormous and doubtless frustrating amount of *the lead developer’s* time and effort, thus far without a single committed patch, or even a consensus as to what it should (or could) do. Rather than continuing to blunder into the future, I think we need to do a reality check

- http://archives.postgresql.org/message-id/20090718160600.GE5172@fetter.org
Retrofitting is Common

• Mandatory access control for Linux
  ‣ Linux Security Modules [Wright et al.,'02]

• TrustedBSD, SEDarwin, sHype, XSM, …

• Secure windowing systems
  ‣ Trusted X, Compartmented-mode workstation, X11/SELinux [Epstein et al.,'90][Berger et al.,'90][Kilpatrick et al.,'03]

• Java Virtual Machine/SELinux [Fletcher,'06]

• IBM Websphere/SELinux [Hocking et al.,'06]

• And more: Apache, PostgreSQL, dbus, gconf, …
Retrofitting Legacy Code

• What if you had to **add security controls** for a legacy program?

Need systematic techniques to retrofit legacy code for security

Legacy code ➔ Retrofitted code

INSECURE ➔ SECURE
Design for Security

- Perhaps retroactive security is the **wrong approach**
  - Too late to get right

- “**Design for security**” from the outset is the goal
  - But, how do we teach programmers to do that?
  - In a practical and time-effective manner

- Design methodologies may vary widely
What is Needed?

- Programs need **multiple security controls**

  ```
  request_loop (client_data, private_data) {
      read(client_passwd, client_req);
      if (necessary ||
          compare_client(client_passwd, private_data))
          access_object(client_req, client_data);
  }
  ```

- Program reads `client_passwd` and `client_req`
- Don’t leak `private_data` used to check passwords
- Control client request’s access to `client_data`
What is Needed?

- Programs need multiple security controls

```c
request_loop (client_data, private_data) {
    read(client_passwd, client_req);
    if (necessary ||
        compare_client(client_passwd, private_data))
        access_object(client_req, client_data);
}
```

- Privilege separation between compare_client and access_object
- Authorization of access_object
- Auditing of execution of unsafe client_req
Past Efforts

• Automated Hook Placement:
  ▸ Assumptions: *Training wheels*
    • (sensitive data types, hook code)
      [Ganapathy et al., 2005, 2006, 2007]
      [Sun et al., 2011, RoleCast 2011]

• Automated Hook Placement 2:
  ▸ Assumptions: *Training wheels*
    • (constraint models of function and security)
      [Harris et al., 2010, 2013]
Security Goals

- Retrofit security controls automatically
  - From “security programs”
- Assist programmers in producing such security programs
  - From code analyses
- Compile such security programs into minimal cost code for enforcing the expected security goals correctly
  - Across security controls
Outline

- Let’s examine the **problem of retrofitting** for security
  - For authorization
- Then explore **other security controls**
  - For privilege separation and auditing
- Then, discuss how to retrofit **across security controls**
  - Step two
We want to generate complete and minimal authorization hook placements mostly-automatically for legacy code


Placement Comparison

• Based on CCS 2012 Method

• X Server:
  ‣ Manual: 201 hooks
  ‣ Automated: 532 hooks

• Postgres:
  ‣ Manual: ~370
  ‣ Automated: 579

What does this mean?
Hook Hoisting

read(pgcSrc->planemask)
read(pgcSrc->fgPixel)
read(pgcSrc->alu)

... read(pgcSrc->bgPixel)
Hook Removal

Resource res = ClientTable[i]

WindowPtr * pWin = (WindowPtr *) res

WindowPtr * pChild = pWin->firstChild->nextSib

pChild->mapped = True

Remove
Relate to Access Control

Access Control Policy: All-or-nothing

\[ \text{op}_1: \text{read}(\text{pgSrc} \rightarrow \text{planemask}) \]

\[ \text{op}_2: \text{read}(\text{pgSrc} \rightarrow \text{fgPixel}) \]

\[ \text{op}_i: \text{read}(\text{pgSrc} \rightarrow \text{bgPixel}) \]

\[ \text{op}_{23}: \text{read}(\text{pgSrc} \rightarrow \text{bgPixel}) \]

Relate to Access Control: All-or-nothing

Policy: All-or-nothing
Authorization Constraints

- \( Allowed(o) \): Subset of subjects in \( U \) that are allowed to perform operation \( o \).

- **Constraint 1:**
  - \( Allowed(o_1) = Allowed(o_2) \), then \( o_1 \) equals \( o_2 \)

- **Constraint 2:**
  - \( Allowed(o_1) \subset Allowed(o_2) \), then \( o_1 \) subsumes \( o_2 \)
Authorization Constraints

- **Allowed(o)**: Subset of subjects in $U$ that are allowed to perform operation $o$.

- **Constraint 1**: 
  - $\text{Allowed}(o_1) = \text{Allowed}(o_2)$, then $o_1$ equals $o_2$

- **Constraint 2**: 
  - $\text{Allowed}(o_1) \subset \text{Allowed}(o_2)$, then $o_1$ subsumes $o_2$

Set of Authorization Constraints limit the access control policies that can be enforced
Why coarser granularity?

\[ \text{op} \ 1: \ \text{read}(\text{pgSrc} \rightarrow \text{planemask}) \]

\[ \text{op} \ 2: \ \text{read}(\text{pgSrc} \rightarrow \text{fgPixel}) \]

\[ \text{op} \ 23: \ \text{read}(\text{pgSrc} \rightarrow \text{bgPixel}) \]

Equivalence:

\[ \text{op}_1 = \text{op}_2 = \ldots = \text{op}_{23} \]
Resource res = clientTable[i]

WindowPtr * pWin = (WindowPtr *) res

Subsumption:

\[ \text{op}_1 > \text{op}_2 \]
\[ \text{op}_3 \not> \text{op}_4 \]

**op\_3:**
PropertyPtr * pProp = pWin->userProps->next

**op\_4:**
pProp->data = data

**op\_1:**
WindowPtr * pChild = pWin -> firstChild -> nextSib

**op\_2:**
pChild->mapped = true
Build Retrofitting Policies

• How do programmers build retrofitting policies?
  ‣ Hundreds of hooks could be removed

•
Build Retrofitting Policies

• However, there are common policy assumptions
  ‣ E.g., object flows – if two operations produce the same data flow, such as from the object to the client (read), then they may be assumed to be equivalent
  ‣ Under this constraint, we could still enforce MLS

• Apply “constraint selectors” to collect such authorization constraints from code
  ‣ Removes up to 2/3 of the unnecessary hooks
Retrofitting for Authorization

- (1) Identify security-sensitive operations
  - Mostly-automated identification of operations [CCS 2012]
- (2) Produce retrofitting policy
  - Produce default authorization hook placement for SSOs
  - Apply constraint selectors for high-level policy constraints
  - Interactive selection of other authorization constraints
- (3) Generate minimal* authorization hook placement
  - Based on retrofitting policy (* modulo assumptions)
- (4) Validate reference monitor concept relative to retrofitting policies and correct transformation
Other Security Controls

• Retrofitting for Privilege Separation and Auditing


Retrofitting for Auditing
Retrofitting for Auditing

• *Audit logs* are intended to provide information about programs to support:
  ‣ Accountability and proof of authorization.
  ‣ Surveillance and intrusion detection.
  ‣ Dynamic analysis for performance/security evaluation.

• Current practice *missing crucial foundational elements*:
  ‣ What is the *formal relation* between a program and its audit log?
  ‣ What *policy* specifies audit log generation?
We propose an information algebraic semantics of auditing that takes as input:

- An arbitrary program `p` in a given language.
- A logging policy `LP` that specifies conditions for logging particular events. (i.e., retrofitting policy)

This semantics, written `genlog(p, LP)` denotes a set of information. An audit log `L` is sound (resp. complete) with respect to the policy iff:

- `L \leq genlog(p, LP)` (resp. `genlog(p, LP) \leq L`) where `\leq` is an information containment relation.
Putting it all together

• Retrofit for multiple security controls
  ‣ **Claim**: reasoning about retrofitting policies across security controls enables benefits
Retrofitting for All

- **Benefits** of retrofitting policies
  - Separate security program from functional program
    - Prevent errors in integration of the two – *even for updates*
    - Make policy enforcement expectations explicit
  - Leverage the relationships between security controls
    - Remove redundant security controls
    - Use security controls to improve retrofitting policies
- **Bottom line**: there is no silver bullet - programmers will need to add such security controls
Summary

• **Problem**: Place Security Controls in Legacy Code
  ‣ Hard to do manually

• **Insights**:
  ‣ Program expectations of security controls into “retrofitting policies” or “security programs”
  ‣ Retrofit programs automatically to minimize cost, validate correctness of security and function
  ‣ Apply across a set of security controls for coherent “Defense in Depth”

• **Targets**: Authorization, Privilege Separation, and Auditing

• **Future**: How shall programmers “Design/program for security”? 