Thoughts on Retrofitting Legacy Code for Security

Somesh Jha
University of Wisconsin

Kaminsky scores 23, No. 5 Wisconsin beats Illinois 68-49
Feb 15, 2015
Threat Landscape:
Summary of Symantec Threat Report 2014
Key Findings

- 91% increase in targeted attacks campaigns in 2013
- 62% increase in the number of breaches in 2013
- Over 552M identities were exposed via breaches in 2013
- 23 zero-day vulnerabilities discovered
- 38% of mobile users have experienced mobile cybercrime in past 12 months
Key Findings (Contd.)

- Spam volume dropped to 66% of all email traffic
- 1 in 392 emails contain a phishing attacks
- Web-based attacks are up 23%
- 1 in 8 legitimate websites have a critical vulnerability
What I feel like?
News is Grim

- See talks at
  - DARPA Cyber Colloquium

- What do we do?
Clean-slate Design

- Rethink the entire system stack

- Networks
  - NSF program
    - See http://cleanslate.stanford.edu
  - See DARPA Mission Resilient Clouds (MRC) program

- Hosts
  - DARPA CRASH program
Some Interesting Systems

- Operating systems with powerful capabilities
  - Asbestos, HiStar, Flume
  - Capsicum
  - ...

- Virtual-machine based
  - Proxos
  - Overshadow

- Possible to build applications with strong guarantees
  - *Web server*: No information flow between threads handling different requests
Two Guiding Principles

- Provide powerful primitives at lower levels in the “system stack”
  - *Example:* HiStar (information flow labels at the OS level)

- Systems will be compromised, but limit the damage
  - *Example:* Process can be compromised, but sensitive data cannot be exfiltrated
What happens to all the code?

- Should we implement all the code from scratch?
- Can we help programmers adapt their code for these new platforms?

Analogy
- We have strong foundation
- Can we build a strong house on top of it?
Ideal Functionality

- **Input**: functionality/security policy
  - **Output**: functional/secure code

- Proving safety is “undecidable”
  - Rice’s theorem (proving any non-trivial property is undecidable)

- I think
  - Synthesis is “relatively hard”
    - Even if provided with an oracle to prove safety
Retrofitting legacy code

Need systematic techniques to retrofit legacy code for security
Premise

- Techniques and ideas from
  - Verification
  - Static Analysis
  - ...

- Can help with this problem
Collaborators and Funding

[Images of collaborators]

[Logos of NSF and DARPA]
The Problem

I DROPPED MY MACBOOK

ON MY OTHER MACBOOK
Rewriting Programs for a Capability System
[Harris et. al., Oakland 2013]

- **Basic problem:** take an insecure program and a policy, instrument program to invoke OS primitives to satisfy the policy

- **Key technique:** reduce to safety game between program and instrumentation
Capsicum
What is Capsicum?

- Capsicum is a lightweight OS capability and sandbox framework developed at the University of Cambridge Computer Laboratory
  - supported by grants from Google, the FreeBSD Foundation, and DARPA.
  - Capsicum extends the POSIX API, providing several new OS primitives to support object-capability security on UNIX-like operating systems:
Capsicum

- [https://www.cl.cam.ac.uk/research/security/capsicum/](https://www.cl.cam.ac.uk/research/security/capsicum/)

- The FreeBSD implementation of Capsicum, developed by Robert Watson and Jonathan Anderson, ships out of the box in FreeBSD 10.0 (and as an optionally compiled feature in FreeBSD 9.0, 9.1, and 9.2)

- Also available on Linux
Running example: gzip

```c
gzip() {
    files = parse_cl;
    for (f in files)
        (in, out) = open;
        compr(in, out);
}

compr(in, out) {
    body;
}
```

public_leak.com
An Informal Policy for gzip

When gzip executes body, it should only be able to read from in and write to out.
Capsicum: An OS with Capabilities

- Two levels of capability:
  - High Capability (can open files)
  - Low Capability (cannot open files)

- Rules describing capability:
  1. Process initially executes with capability of its parent
  2. Process can invoke the drop system call to take Low Capability
Securing gzip on Capsicum

gzip() {
    files = parse_cl;
    for (f in files)
        (in, out) = open;
    compr(in, out);
}

compr(in, out) {
    drop();
    body; 
    public_leak.com
}

High Cap.

Low Cap.
Securing gzip on Capsicum

gzip() {
    files = parse_cl;
    for (f in files) {
        (in, out) = open;
        compr(in, out);
    }
}

compr(in, out) {
    drop();
    body;
}

Low Cap.
Securing gzip on Capsicum

gzip() {
    files = parse_cl;
    for (f in files) {
        (in, out) = open;
        compr(in, out);
    }
}

compr(in, out) {
    drop();
    body;
}

Low Cap. ≠ High Cap.
Securing gzip on Capsicum

gzip() {
    files = parse_cl;
    for (f in files) {
        (in, out) = open;
        compr(in, out) { 
            drop();
            body;  
        }
        fork(_compr(in, out));
    }
}
Securing gzip on Capsicum

gzip() {
    files = parse_cl;
    for (f in files)
        (in, out) = open;  // High Cap.
        fork_c(compr(in, out));
}

cmpr(in, out) {
    drop();
    body;
    Low Cap.
}
Insecure Program

gzip() {
    ...
    compr();
    ...
}
compr(...) { ... }

gzip should always execute comp() with low cap, but always open files in main with high cap

Secure Program

gzip() {
    ...
    fork_compr();
    ...
}
compr(...) {
    drop();
    ...
}
Policies are not *instrumented programs*, and they should be explicit.
First Key Insight

Insecure Program

gzip() {
  ...
  compr();
  ...
}
compr(...) { ... }

Disallowed Executions
  .* [ compr() with high cap ]
  | .* [ open() with low cap ]

Secure Program

gzip() {
  ...
  fork_compr();
  ...
}
compr(...) { ...
  drop();
  ...
}

gzip should always execute compr() with low cap, but always open files in main with high cap.
Second Key Insight

From an insecure program and policy, we can automatically write a secure program by solving a two-player safety game.

[Harris et. al., CAV 2012]
Second Key Insight

Insecure Program
```
gzip() {
    ...
    compr();
    ...
} compr(...) { ... }
```

Disallowed Executions
```
disallowed executions
| .* [ compr() with high cap ]
| .* [ open() with low cap ]
```

Secure Program
```
gzip() {
    ...
    fork_compr();
    ...
} compr(...) {
    drop();
    ...
}
The Technique

HOW IT WORKS

1. Cap containing a figure-of-eight shaped magnet connected to an electric current is placed on head. Magnet is made up of a bundle of intertwined wires and is near the left ear.

2. The tiny magnetic pulses disturb electric circuits on left side of the brain, which usually sees the ‘bigger picture’ and suppresses the detail-hoarding right side.

3. Details filed unconsciously come to the fore, creating a burst of creative, mathematical or other talent.
Weaving as a Game

Two steps:

1. Model uninstrumented program, policy, and Capsicum as languages/automata

2. From automata, translate weaving problem to a two-player safety game
Step 1: Model

- **Program** is a language over program instructions (Instrs)

- **Policy** is a language of instructions paired with capability (Instrs x Caps)

- Capsicum is a *transducer* from instructions and primitives to capabilities (Instrs U Prims → Caps)
Step 2: Construct a Game

- From models, construct a “game” between insecure program and instrumentation

- Program plays instructions (Instrs), instrumentation plays primitives (Prims)

- Program wins if it generates an execution that violates the policy
Safety Games: A Primer

Two players: **Attacker** and **Defender**

Play: **Attacker** and **Defender** choose actions in alternation

Player goals:
- **Attacker**: generate a play accepted by the game
- **Defender**: thwart the **Attacker**
Winning Strategy

Winning strategy: choices that a player can make to always win a game
Some Details
Experimental Highlights

• capweave policies are small compared to program size (10’s of lines vs. thousands)

• capweave instruments most programs fast enough to be in an edit-compile-run cycle

• capweave-rewritten programs have reasonable overhead vs. hand-rewritten
## capweave Performance

<table>
<thead>
<tr>
<th>Name</th>
<th>kLoC</th>
<th>Pol. Lines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>8</td>
<td>70</td>
<td>4m57s</td>
</tr>
<tr>
<td>gzip</td>
<td>9</td>
<td>68</td>
<td>3m26s</td>
</tr>
<tr>
<td>php-cgi</td>
<td>852</td>
<td>114</td>
<td>46m36s</td>
</tr>
<tr>
<td>tar</td>
<td>108</td>
<td>49</td>
<td>0m08s</td>
</tr>
<tr>
<td>tcpdump</td>
<td>87</td>
<td>52</td>
<td>0m09s</td>
</tr>
<tr>
<td>wget</td>
<td>64</td>
<td>35</td>
<td>0m10s</td>
</tr>
</tbody>
</table>
# Weaved-program Performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Tests</th>
<th>Passed</th>
<th>Overhead: capweave / hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>6</td>
<td>6</td>
<td>20.90%</td>
</tr>
<tr>
<td>gzip</td>
<td>2</td>
<td>2</td>
<td>15.03%</td>
</tr>
<tr>
<td>php-cgi</td>
<td>11</td>
<td>2</td>
<td>65.64%</td>
</tr>
<tr>
<td>tar</td>
<td>1</td>
<td>1</td>
<td>64.78%</td>
</tr>
<tr>
<td>tcpdump</td>
<td>29</td>
<td>27</td>
<td>24.77%</td>
</tr>
<tr>
<td>wget</td>
<td>4</td>
<td>4</td>
<td>0.91%</td>
</tr>
</tbody>
</table>
Additional Challenges

• User Study
  – Patterson: “How do you know you are doing better?”

• Optimizations
  – Incorporate quantitative measures into games (e.g., mean-payoff games)

• User-friendliness
  – Better policy language
The Future
OK... but when does it end?

Decentralized Information Flow
- Asbestos [SOSP 2005]
- HiStar [SOSP 2006]
- Flume [SOSP 2007]*

Analogous problem to capabilities
- Capabilities $\approx$ flows
- drop $\approx$ labels

* Related work in [Harris et. al., CCS 2010]
Program → Policy → hiweave → HiStar → Weaver Generator

HiStar Program
Questions
Summary

Insecure Program:
gzip() {
    ...
    compr();
    ...
} 
cmpr(...) { ... }

Secure Program:
gzip() {
    ...
    fork_compr();
    ...
} 
cmpr(...) {
    drop();
    ...
} 

Disallow Executions:
.* [ compr() with high cap ]
| .* [ open() with low cap ]

capweave