Efficient Patch-based Auditing for Web Application Vulnerabilities

Taesoo Kim, Ramesh Chandra, Nickolai Zeldovich

MIT CSAIL
Example: Github

- Github hosts projects (git repository)
- Users have own projects
- **Authentication** based on SSH public key
Vulnerability: attacker can modify any user's public key

- Publicly announced in March 2012
- **Unauthorized user** modified Ruby-on-Rails project after **modifying** a developer's **public key**.

```
PUBLIC rails / rails

wow how come i commit in master? O_o

homakov authored 3 months ago

Showing 1 changed file with 3 additions and 0 deletions.

+ hacked

hacked

... @@ -0,0 +1,3 @@
1 +another showcase of rails apps vulnerability.
```
Problem: who exploited this vulnerability?

- Other attackers may have known about the vulnerability **for months or years**
- Adversaries could have modified many users' public keys, repositories, etc.
- **Ideally**, would like to detect **past attacks** that exploited this vulnerability
Github's actual response

- Immediately **blocked** all users
- **Asked** users to **audit** own public key

---

Action Required - SSH Key Vulnerability

GitHub 📧 support@github.com  Mar 7 ★

to me ➜

#Required Action

Since you have one or more SSH keys associated with your GitHub account, you must visit [https://github.com/settings/ssh/audit](https://github.com/settings/ssh/audit) to approve each valid SSH key.

Until you have approved your SSH keys, you will be unable to clone/pull/push your repositories over SSH.
Detecting past attacks is hard

- Current tools require **manual** log analysis
- Logs may be **incomplete**
- Logs may be **large** (Github: 18M req/day)
Too many vulnerabilities to inspect manually

- CVE database: 4,000 vulnerabilities per year
- Hard enough for administrator to apply patches
- Auditing each vulnerability for past attacks is impractical
Approach: automate auditing using patches

- **Insight**: security patch renders attack harmless

- **Technique**: compare execution of each request before and after patch is applied
  - Same result: no attack
  - Different results: potential attack!
Example: Github vulnerability

```
<form>
  <input type="text" name="key">
  <input type="hidden" value="taesoo" name="id">
</form>
```
Example: Github vulnerability

```ruby
params = {
  "key" => "ssh-rsa AAA ... ",
  "id" => "taesoo"
}

def update_pubkey
  @key = PublicKey.find_by_id(params['id'])
  @key.update_attributes(params['key'])
end
```
def update_pubkey
    @key = PublicKey.find_by_id(params['id'])
    @key.update_attributes(params['key'])
end
Example: Github vulnerability

```ruby
def update_pubkey
  @key = PublicKey.find_by_id("victim")
  @key.update_attributes("attacker's public key")
end
```

```ruby
def update_pubkey
  @key = PublicKey.find_by_id("victim")
  @key.update_attributes("attacker's public key")
end
```

Attackers can **overwrite** any user's public key, and thus can **modify** user's repositories.
def update_pubkey
    @key = PublicKey.find_by_id(params['id'])
    @key = PublicKey.find_by_id(cur_user.id)
    @key.update_attributes(params['key'])
end

Login-ed user's id
Patch-based auditing finds attack

- **Replay** each request using old(-) & new(+) code
- Attack request generates different SQL queries

```ruby
def update_pubkey
  - @key = PublicKey.find_by_id(params['id'])
  + @key = PublicKey.find_by_id(cur_user.id)
  @key.update_attributes(params['key'])
end
```

- UPDATE ... WHERE KEY=... ID=victim
- UPDATE ... WHERE KEY=... ID=attacker
Challenge: auditing many requests

- Necessary to audit huge amount of requests
  - Vulnerability may have existed for a long time
  - Busy web applications may have many requests
    (Github: 18M req/day)

- Auditing one month traffic requires two months
  - Naive approach requires two re-executions
    (old & new code) per request
Contribution

- **Efficient** patch-based auditing for web apps.

- *12 – 51x* faster than original execution for challenging patches
  - Worst case, auditing one month worth of requests takes 14 – 60 hours
Overview of design

Runtime
- HTTPD
- PHP

Auditing
- Audit Ctrl
- Replayer

Audit log

patch
suspect requests
Admin
Logging during normal execution

- CGI, GET, POST ...
  - initial input
- PHP
- rand()
  - non-deterministic input
- mysql_query()
  - external input
- HTML
Auditing a request

PHP

rand() → mysql_query() → HTML

original

HTML → compare?

Auditing

PHP

rand() → mysql_query() → HTML

patched
Auditing a request

Naive approach requires **two complete** re-executions for **every** request

**Auditing**
Opportunities to improve auditing performance

• Patch might **not affect** every request
  • How to determine affected requests?

• Original and patched runs execute **common code**
  • How to **share** common code during re-execution?

• **Multiple requests** execute similar code
  • How to **reuse** similar code across **multiple** requests?
Key ideas

• Idea 1: Control flow filtering
  • Auditing only affected requests

• Idea 2: Function-level auditing
  • Sharing common code during re-execution

• Idea 3: Memoized re-execution
  • Reusing memoized code across multiple requests
Idea 1: Control flow filtering

- Step 1: Normal execution
  - Record the control flow trace (CFT) of each request

- Step 2: Indexing
  - Map the control flow trace (CFT) to the basic blocks

- Step 3: Auditing
  - Compute the basic blocks modified by the patch
  - Filter out requests if did not execute any patched basic blocks
Static analysis of source code

- Computing **basic blocks** of source code

```php
function get_name() {
    return $_GET['name'];
}

if ($_GET['q'] == 'echo') {
    echo get_name();
}
```
Static analysis of source code

- Computing **basic blocks** of source code

```php
function get_name() {
    return $_GET['name'];
}
```

```php
if ($_GET['q'] == 'echo') {
    echo get_name();
}
```

JMP,BRK ...
Recording control flow trace

- Normal execution:

  logging control flow trace (CFT) of each request

/s.php?q=test

```php
function get_name() {
    return $_GET['name'];
}
if ($_GET['q'] == 'echo') {
    echo get_name();
}
start

CFT: [④, ⑥] (file, scope, func, #instruction)
```
Computing executed basic blocks

- Indexing:
  computing **executed basic blocks** of each request

Basic Blocks

1. function get_name() {
2. return $_GET['name'];
3. }
4. if ($_GET['q'] == 'echo') {
5. echo get_name();
6. }

/s.php?q=test
Computing modified basic blocks

- Auditing:
  compute the basic blocks **modified** by the patch

```
function get_name() {
    return $_GET['name'];
    return sanitize($_GET['name']);
}
```

- Basic Blocks

  - [①, ②, ③]
  - [④]
  - [⑤]
  - [⑥]
Comparing basic blocks

- Auditing:
  
  **filter out** the requests that did not execute patched basic blocks

<table>
<thead>
<tr>
<th>Executed</th>
<th>Patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1, 2, 3])</td>
<td>(\checkmark) ([1, 2, 3])</td>
</tr>
<tr>
<td>(\checkmark) ([4])</td>
<td>([4])</td>
</tr>
<tr>
<td>([5])</td>
<td>([5])</td>
</tr>
<tr>
<td>(\checkmark) ([6])</td>
<td>([6])</td>
</tr>
</tbody>
</table>
Summary: control flow filtering
Idea 2: Function-level auditing

- Optimization 1: sharing common code
  - **Share code** up to the patched function

- Optimization 2: early termination
  - **Stop** after the last invocation of the patched functions
Function-level auditing

- **Intercept** side-effects **inside** the patched functions
- **Stop** after the **last** invocation of the patched functions
- **Compare** intercepted **side-effects**

**Auditing**

```
Auditing

fork()

PHP

[compare side-effects?]

original function
patched function
```
Intercepting side-effects

```php
class PublicKey {
    ...
    function update($key) {
        $this->last = date();
        echo "updated";
        $rtn = mysql_query("UPDATE ... $key ...");
        return $rtn;
    }
    ...
}
```

<the worst case example>
Comparing side-effects

- If different, mark the request suspect
- If same, stop and audit next request
Summary: function-level auditing

- Affected requests
- Naive auditing
- Function-level auditing
Idea 3: Memoized re-execution

- **Motivation**: many requests run similar code

```php
function get_name() {
    return $_GET['name'];
}

if ($_GET['q'] == 'echo') {
    echo get_name();
}
```

```
1)/s.php?q=echo&name=alice
```

CFT: [④,⑤,①,②,③,⑥]
Idea 3: Memoized re-execution

• Motivation: many requests run similar code

```php
function get_name() {
    return $_GET['name'];
}
```

```javascript
start if ($_GET['q'] == 'echo') {
    echo get_name();
}
```
Idea 3: Memoized re-execution

- **Motivation**: many requests run similar code

Control flow group (CFG)

```
1) /s.php?q=echo&name=alice
2) /s.php?q=echo&name=bob
3) /s.php?q=echo&name=<script>…
```

```
function get_name() {
    return $_GET['name'];
}
```

```
if ($_GET['q'] == 'echo') {
    echo get_name();
}
```
Idea 3: Memoized re-execution

• Step 1: normal execution
  • **Record** control flow trace (CFT) of each request
  • **Classify** the corresponding control flow group (CFG)

• Step 2: auditing (each CFG)
  • Determine input differences among requests (template variables)
  • Generate a **template**: efficient way to re-execute request given an assignment of template variables
  • **Re-execute** each request using the template
Determining template variables

- **Template variables** are input differences among all requests in the same CFG (e.g., GET/POST, CGI variables, ...)

1) /s.php?q=echo&name=alice
2) /s.php?q=echo&name=bob
3) /s.php?q=echo&name=<script> ... 

(e.g., $GET[name] = Template variable)
Generating a template

Template variable

/s.php?q=echo&name=alice

1. function get_name() {
2. return $_GET['name'];
3. }

4. if ($_GET['q'] == 'echo') {
5. echo get_name();
6. }

→ Template: [②, ⑤]

1. Determine template variables of the CFG
2. Pick / replay a request from the CFG
3. Record instructions depending on template variables
Re-executing the template

1) /s.php?q=echo&name=alice
2) /s.php?q=echo&name=bob
3) /s.php?q=echo&name=<script> ...

1. Update the template variable
   (e.g., $_GET['name'] = 'bob' and '<script>...')

2. Re-execute the recorded instructions in the template

  ② return $_GET['name'];
  ⑤ echo return of ②;
Auditing with template re-execution

3)/s.php?q=echo&name=<script> ...

1. Given a patch

2. Re-execute the template up to the patched function

3. Perform function-level auditing
Summary: template re-execution

Affected requests

CFG

CFG

Template re-execution

Template
Optimization: collapsing templates

- **Motivation**: different CFGs can share common code up to the patched function (given patch)
Summary: collapsing template

Template re-execution

CFG

CCFG

Auditing

Template
Implementation

- **POIROT**: a prototype for PHP
  - Based on PHP-5.3.6
  - Using PHP Vulcan Logic Dumper
  - 15,000 LoC changes

- **No changes** in application source code
Evaluation

- Does POIROT detect attacks of real vulnerabilities?
- Does POIROT audit efficiently?
- Does POIROT impose reasonable runtime overhead?
POIROT detects real attacks

- **MediaWiki**: detected 5 different types of attacks (using *realistic* Wikipedia traces)
- **HotCRP**: detected 4 information leak vulnerabilities (using *synthetic* workloads)

<table>
<thead>
<tr>
<th>CVE</th>
<th>Description</th>
<th>Detected?</th>
<th>F+</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-4589</td>
<td>Stored XSS</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>2009-0737</td>
<td>Reflected XSS</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>2010-1150</td>
<td>CSRF</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>2004-2186</td>
<td>SQL injection</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>2011-0003</td>
<td>Clickjacking</td>
<td>Yes</td>
<td>100%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUG</th>
<th>Detected?</th>
<th>F+</th>
</tr>
</thead>
<tbody>
<tr>
<td>f30eb</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>63896</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>3ff7b</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>4fb7d</td>
<td>Yes</td>
<td>0</td>
</tr>
</tbody>
</table>

**MediaWiki**

**HotCRP**
POIROT efficiently audits attacks

- **34 CVEs** (security patches 2004 ~ 2011)
- Trace containing 100K Wikipedia requests (3.4 h)
- Auditing time:
  - 29 CVEs: <0.2 sec
  - 5 CVEs: ~9.2 min (12x ~ 51x faster than the original execution)

<table>
<thead>
<tr>
<th>CVE</th>
<th>Naive Time (h)</th>
<th>POIROT Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0537</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0003</td>
<td>7.0 h</td>
<td>16.5 min</td>
</tr>
<tr>
<td>2007-1055</td>
<td>6.8 h</td>
<td>16.9 min</td>
</tr>
<tr>
<td>2007-0894</td>
<td>8.8 h</td>
<td>4.0 min</td>
</tr>
<tr>
<td><strong>29 cases</strong>*</td>
<td><strong>6.9 h</strong></td>
<td><strong>0.02~0.19 s</strong></td>
</tr>
</tbody>
</table>

* 2011-1766, 2010-1647, 2011-1765, 2011-1587, ...
Control flow filtering is effective for many patches

- **34 CVEs** (security patches 2004 ~ 2011)
- Trace containing **100K** Wikipedia requests (3.4 h)
- Auditing time:
  - **29 CVEs**: <0.2 sec
  - **5 CVEs**: ~9.2 min (12x ~ 51x faster than the original execution)

<table>
<thead>
<tr>
<th>CVE</th>
<th>Naive Time (h)</th>
<th>POIROT Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0537</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0003</td>
<td>7.0 h</td>
<td>16.5 min</td>
</tr>
<tr>
<td>2007-1055</td>
<td>6.8 h</td>
<td>16.9 min</td>
</tr>
<tr>
<td>2007-0894</td>
<td>8.8 h</td>
<td>4.0 min</td>
</tr>
<tr>
<td><strong>29 cases</strong></td>
<td><strong>6.9 h</strong></td>
<td><strong>0.02~0.19 s</strong></td>
</tr>
</tbody>
</table>

Control flow filtering is effective for many patches

- 34 CVEs (security patches 2004 ~ 2011)
- Trace containing 100K Wikipedia requests (3.4 h)
- Auditing time:
  - 29 CVEs: <0.2 sec
  - 5 CVEs: ~9.2 min (12x ~ 51x faster than the original execution)

<table>
<thead>
<tr>
<th>CVE</th>
<th>Naive Time (h)</th>
<th>POIROT Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0537</td>
<td>6.6 h</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0003</td>
<td>7.0 h</td>
<td>16.5 min</td>
</tr>
<tr>
<td>2007-1055</td>
<td>6.8 h</td>
<td>16.9 min</td>
</tr>
<tr>
<td>2007-0894</td>
<td>8.8 h</td>
<td>4.0 min</td>
</tr>
<tr>
<td>29 cases*</td>
<td>6.9 h</td>
<td>0.02~0.19 s</td>
</tr>
</tbody>
</table>

Function-level auditing improves performance

- Naive: 7.3 h → Func-level: 3.5 h
- Re-execute 2 – 60% (avg. 16%) instructions

<table>
<thead>
<tr>
<th>CVE</th>
<th>#re-exec. Instructions / #total instructions</th>
<th>Func-level Re-exec (hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>6.4K / ~200K = 3.2%</td>
<td>2.4 h</td>
</tr>
<tr>
<td>2011-0537</td>
<td>4.8K / ~200K = 2.4%</td>
<td>5.3 h</td>
</tr>
<tr>
<td>2011-0003</td>
<td>120K / ~200K = 58.5%</td>
<td>5.4 h</td>
</tr>
<tr>
<td>2007-1055</td>
<td>5.6K / ~200K = 2.79%</td>
<td>2.0 h</td>
</tr>
<tr>
<td>2007-0894</td>
<td>25K / ~200K = 12.5%</td>
<td>2.9 h</td>
</tr>
</tbody>
</table>
Templates reduce re-executed instructions

- 100K requests $\rightarrow$ ~840 #CFG
- Templates contain 0.1% ~ 2.7% (avg. 0.7%) instruction

<table>
<thead>
<tr>
<th>CVE</th>
<th>#CFG</th>
<th>#instruction in a template / #total instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>844</td>
<td>289 / 200K = 0.14%</td>
</tr>
<tr>
<td>2011-0537</td>
<td>834</td>
<td>96 / 200K = 0.05%</td>
</tr>
<tr>
<td>2011-0003</td>
<td>834</td>
<td>5,427 / 200K = 2.71%</td>
</tr>
<tr>
<td>2007-1055</td>
<td>844</td>
<td>177 / 200K = 0.09%</td>
</tr>
<tr>
<td>2007-0894</td>
<td>844</td>
<td>1,085 / 200K = 0.54%</td>
</tr>
</tbody>
</table>
Collapsing reduces number of templates

- 100K → ~840 #CFG → 1 ~ 589 #CCFG
- 30.5 s to collapse templates on average
- Auditing 100K requests (3.4h) → avg. 9.2 min

<table>
<thead>
<tr>
<th>CVE</th>
<th>#CCFG / #CFG</th>
<th>Collapsing time (sec)</th>
<th>Memoized POIROT (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011-4360</td>
<td>4 / 844 = 0.5%</td>
<td>31.0</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0537</td>
<td>1 / 834 = 0.1%</td>
<td>30.3</td>
<td>4.5 min</td>
</tr>
<tr>
<td>2011-0003</td>
<td>589 / 834 = 69.8%</td>
<td>30.5</td>
<td>16.5 min</td>
</tr>
<tr>
<td>2007-1055</td>
<td>2 / 844 = 0.2%</td>
<td>30.1</td>
<td>16.9 min</td>
</tr>
<tr>
<td>2007-0894</td>
<td>18 / 844 = 2.1%</td>
<td>30.4</td>
<td>4.0 min</td>
</tr>
</tbody>
</table>
POIROT imposes moderate runtime overhead

- Testing with 100K Wikipedia requests
  - 14.1% latency overhead
  - 15.3% throughput overhead
  - 5.4 KB/req storage overhead (compressed online)
Related work

- Record-and-replay with patches:
  - **Warp**: repairing web apps with retroactive patching
  - **Rad**: fork-and-compare, auditing memory writes

- Testing patched programs:
  - **TACHYON**: automatic/live patch testing
  - **Delta execution**: validate patched version (split/merge)

- Program slicing (adjustable computation):
  - **Static slicing**: all stmts. that possibly affect the variable
  - **Dynamic slicing**: all stmts. that really affected the variable
Conclusion

- POIROT: efficient patch-based auditing system
  - Detected real attacks in MediaWiki / HotCRP without any modification
  - 12 – 51x faster than original execution

- Three partial re-execution techniques
  - Control flow filtering
  - Function-level auditing
  - Memoized re-execution