Devin Cook
Yung Ryn Choe
Drew Hamilton
Auburn University
Sandia National Laboratories
CSET 2015

General Problem Statement

How it Works

Results

Conclusions

Background

Proposal

Literature Review

Finding Bugs

The Idea

Finding Bugs in Source Code Using Commonly Available Development Metadata
Devin Cook
Yung Ryn Choe
Drew Hamilton
Auburn University
Sandia National Laboratories
CSET 2015

General Problem Statement
Finding Bugs in Source Code Using Commonly Available Development Metadata

Results
Conclusions

Literature Review
Background

The Idea
Proposal

How it Works

Prezi
General Problem Statement

**Goal:** Find bugs before they are able to be exploited.

Everyone wins!*  
*except the bad guys

So you've found an 0day... what next?

- Sell it.  
- Use it for good.  
- Disclose it.  
- Use it for evil.

**Central Hypothesis**
We can use development metadata from a specific project (namely bug reports and changeset information) to find new bugs in that project and others that are difficult to find using traditional static analysis techniques alone.
Anatomy of an Exploit
Goal:
Find bugs before they are able to be exploited.

Everyone wins!*  
*except the bad guys
So you've found an 0day... what next?

Sell it.
Use it for good.

Disclose it.
Use it for evil.
Central Hypothesis

We can use development metadata from a specific project (namely bug reports and changeset information) to find new bugs in that project and others that are difficult to find using traditional static analysis techniques alone.
Devlin Cook  
Yung Ryn Choe  
Drew Hamilton  
Auburn University  
Sandia National Laboratories  
CSET 2015

General Problem Statement

Finding Bugs in Source Code Using Commonly Available Development Metadata

Literature Review

The Idea

Proposal

How it Works

Results

Conclusions

Background

Prezi
Finding Bugs

- Fuzzing
- Dynamic Analysis
- Static Analysis
Typical Workflow

Source Code ➔ Tool ➔ Rules/Plugins ➔ Potential Vulnerabilities
Common Themes:
- Modular (front end/back end)
- Some kind of AST/IR
- Dawn Song and David Brumley

MAYHEM

Early standard detection techniques although optimized, but generates working exploits.
Literature Review

Finding Bugs

Typical Workflow

Using Development Metadata

Matching ASTs

BugBench
The Idea

Direct Benefits
More secure code faster.

Value Adds
- Detection of vulnerabilities in code.
- Integration with existing tools.
- Improved security through continuous monitoring.

Proposed Architecture

Proposed Methodology

Core tasks:
- Identify that a piece of code relates to a vulnerability.
- Collect all related data.
- Use that new information to detect other vulnerabilities.
Explo(it)ring the WordPress Extension Repos

Today's post is kind of long, so I thought I should warn you in advance by adding an additional paragraph for you to read. I also wanted to provide download links for those who'd rather just read the code. It isn't the cleanest code in the world, so I apologize in advance. I discuss what all of these are for and how they work later on in the post, so if you're confused and/or curious, read on.

Downloads:

- Copies of the WordPress theme and plugin repositories can be grabbed via torrent (Please note that the plugin repo has a few directories incomplete/missing; this can be fixed by running my checkout code)
- A new WordPress plugin fingerprinting tool, wpfinger (download). This tool can infer detailed version information on just about every plugin in the WordPress repository. This package also contains some useful libraries for checking out the repositories and scraping plugin rankings, as this is used in the fingerprinting tool.

Intro

After finding an arbitrary file upload vulnerability in 1 Flash Gallery, I became curious as to how many other WordPress plugins made basic security mistakes. The 1 Flash Gallery plugin issue, it seems, is that they CTRL-C-V'd code from a project called Uploadify, which has been known to be vulnerable for quite awhile.

After realizing this, I became curious as to how many plugins make easy-to-spot security mistakes, such as reusing vulnerable libraries or doing such things as include($_REQUEST['lulz']). However, my curiosity was initially somewhat hampered by the fact that downloading and auditing every WordPress plugin one at a time is not only a mind numbing task, but a herculean one as well. And, well, I'm incredibly lazy.

Getting the Repos

So what do? Well, it turns out that WordPress is nice enough to have public repositories (http://plugins.svn.wordpress.org and http://themes.svn.wordpress.org) containing all plugins that have ever been submitted, as well as every theme. This, of course, was exciting: I could just check this out, whip out some grep-fu, and have my answers.
WordPress Plugin Repository
WordPress Plugin Repository

```
$ egrep -i '(include|required)(\_once)?\((\|s+)[^;]*\$\_\(REQUEST\|GET\|POST\|COOKIE\)'
= 11 unauthenticated RFI vulnerabilities
...little to no effort
Proposed Methodology

Core tasks:
- Identify that a piece of data relates to a vuln
- correlate all related data
- use that new information to detect other vulns

Details
- Use additional data as input - more "signals"
  - bugtrackers
  - commit messages
  - mailing lists
  - public vuln DBs (known-vuln code/packages)
- Find patterns
  - certain committers that make certain mistakes
  - certain files or areas that often contain bugs
  - vuln code reuse
  - more vulns late at night?
  - devs with short names introduce more vulns? - could be anything!

Generate a DB of vulnerable patterns/code
- Look for bugs
  - Given a new commit, compute the "security score"
  - Given a new repo (or just some source), go find some 0days!
Proposed Methodology

Core tasks:
- Identify that a piece of data relates to a vuln
- Correlate all related data
- Use that new information to detect other vulns

Details
- Use additional data as input - more "signals"
  - bug trackers
  - commit messages
  - mailing lists
  - public vuln DBs (known vuln code/packages)
- Find patterns
  - Certain committers that make certain mistakes
- Generate DB of vulnerable patterns/code
  - Look for bugs
    - Given a new commit, compute the "security score"
    - Given a new repo (or just some source), go find some 0days!
Final Architecture

1. Data gathering and organization
   - Find new bugs
     - find similar code via text matching
     - find similar commits via machine learning

2. Bug correlation
   - search issues for commit references
   - search commits for issue references

3. Bug injection identification
   - use VCS annotation
   - use test cases and VCS bisection

4. Manual verification
1. Data gathering and organization

**tracker_bugs**
- id integer
- status integer
- severity integer
- title text
- assignee integer
- superseder integer
- bug_type integer
- stage integer
- resolution integer
- priority integer
- reporter integer

**tracker_messages**
- bug integer
- author integer
- message text
- date datetime

---

**Mercurial > cpython**

**changeset 86676:10d0edadbcdd 3.3**

Issue #17997: Change behavior of `ssl.match_hostname()` to follow RFC 3433 for security reasons. It now doesn't match multiple wildcards nor wildcards inside IDN fragments. [#17997]

author Georg Brandl <georg@python.org>
date Sun, 27 Oct 2013 07:16:53 +0100 (14 months ago)
2. Bug correlation
   - search issues for commit references
   - search commits for issue references
3. Bug injection identification
   • use VCS annotation
   • use test cases and VCS bisection

$ hg blame ssl.py
...
65179: def match_hostname(cert, hostname):
   ...
65179:     if not cert:
87688:       raise ValueError("empty or no certificate, match_hostname needs a 
87688:       "SSL socket or SSL context with either 
87688:       "CERT_OPTIONAL or CERT_REQUIRED")
65179:     dnsnames = []
65179:     san = cert.get('subjectAltName', ())
65179:     for key, value in san:
65179:       if key == 'DNS':
86676:         if _dnsname_match(value, hostname):
65179:           return
65179:     dnsnames.append(value)
...

--- a/Lib/test/test_ssl.py
+++ b/Lib/test/test_ssl.py
...
+ def test_parse_cert_CVE_2013_4073(self):
+   p = ssl._ssl._test_decode_cert(NULLBYTECERT)
+   if support.verbose:
+     sys.stdout.write("\n" + pprint.pformat(p) + "\n")
+   subject = ("('countryName', 'US'),"),
+     ("('stateOrProvinceName', 'Oregon'),"),

4. Find new bugs
   • find similar code via text match
4. Find new bugs
   - find similar code via text matching
   - find similar commits via machine learning
1. Data gathering and organization

- **tracker_bugs**
  - id integer
  - status integer
  - severity integer
  - title text
  - assignee integer
  - supersede integer
  - bug_type integer
  - stage integer
  - resolution integer
  - priority integer
  - reporter integer

- **tracker_messages**
  - bug integer
  - author integer
  - message text
  - date datetime

2. Bug correlation

- search issues for commit references
- search commits for issue references

3. Bug injection identification

- use VCS annotation
- use test cases and VCS bisection

4. Find new bugs

- find similar code via text matching
- find similar commits via machine learning

5. Manual verification
Results

Test Sets

Python
- Large codebase
- Mostly C and Python
- Routine Cverity scans
- 93,000+ commits
- 192 committers
- ~1M lines of code
- 34,000 issues
- 162 security issues

Roundup
- Order of magnitude smaller
- Mostly Python
- <5,000 commits
- ~100K lines of code
- 22 security issues

Validation
- Can we find bugs?
- Is the false positive rate reasonable?
- Cverity developers say <50%
- Is the false negative rate reasonable?

Development Metrics

Conclusions
- Novel approach for identifying vulnerability injection
- Great at finding new bugs that are similar to bugs that were already fixed and can sometimes revolve bugs
- Can still find bugs in projects with insufficient development metadata
- Almost entirely automated
Test Sets

- Large codebase
- Mostly C and Python
- Routine Coverity scans

93,000+ commits
192 committers
~1M lines of code
34,000 issues
162 security issues

Python:

Uns in reference WSGI implementation
Information disclosure via pydoc server

Roundup

- Order of magnitude smaller
- Mostly Python

<5,000 commits
~100K lines of code
22 security issues
- Large codebase
- Mostly C and Python
- Routine Coverity scans

93,000+ commits
192 committers
~1M lines of code
34,000 issues
162 security issues
Roundup

- Order of magnitude smaller
- Mostly Python

<5,000 commits
~100K lines of code

22 security issues
Python:
  - DoS in reference WSGI implementation
  - Information disclosure via pydoc server

Roundup:
  - 2x Unsafe Python functions
  - 2 others were mitigated directory traversal vulnerabilities
[*] issue10714 {closed | security | msg: 4 | related: 3}

   self.raw_requestline = self.rfile.readline()

cpython/Lib/wsgiref/simple_server.py: self.raw_requestline = self.rfile.readline()

[*] issue672656 {closed | security | msg: 5 | related: 2}

   self.address = ('', port)

cpython/Lib/pydoc.py: self.address = ('', port)
Validation

- Can we find 0days?
- Is the false positive rate reasonable?
  - Coverity developers say <30%
- Is the false negative rate reasonable?

<table>
<thead>
<tr>
<th>Project</th>
<th>LOC (Approx.)</th>
<th>Total Issues</th>
<th>Code Signatures</th>
<th>Candidates</th>
<th>True Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>976,413</td>
<td>162</td>
<td>38</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>Roundup</td>
<td>88,578</td>
<td>22</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.5: Vulnerability Detection Rates
Development Metrics

Table 4.1: Python Developer Vulnerability Injection Rate (Top 10)

<table>
<thead>
<tr>
<th>Developer</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev00</td>
<td>14.29%</td>
</tr>
<tr>
<td>Dev01</td>
<td>4.35%</td>
</tr>
<tr>
<td>Dev02</td>
<td>3.23%</td>
</tr>
<tr>
<td>Dev03</td>
<td>2.78%</td>
</tr>
<tr>
<td>Dev04</td>
<td>2.33%</td>
</tr>
<tr>
<td>Dev05</td>
<td>2.08%</td>
</tr>
<tr>
<td>Dev06</td>
<td>1.59%</td>
</tr>
<tr>
<td>Dev07</td>
<td>1.59%</td>
</tr>
<tr>
<td>Dev08</td>
<td>1.19%</td>
</tr>
<tr>
<td>Dev09</td>
<td>1.15%</td>
</tr>
</tbody>
</table>

Table 4.2: Python Developer Vulnerability Reports (Top 10)

<table>
<thead>
<tr>
<th>Developer</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev12</td>
<td>18</td>
</tr>
<tr>
<td>Dev44</td>
<td>10</td>
</tr>
<tr>
<td>Dev18</td>
<td>5</td>
</tr>
<tr>
<td>Dev45</td>
<td>3</td>
</tr>
<tr>
<td>Dev46</td>
<td>3</td>
</tr>
<tr>
<td>Dev47</td>
<td>3</td>
</tr>
<tr>
<td>Dev14</td>
<td>3</td>
</tr>
<tr>
<td>Dev24</td>
<td>3</td>
</tr>
<tr>
<td>Dev48</td>
<td>2</td>
</tr>
<tr>
<td>Dev30</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Python Developer Vulnerability Resolutions (Top 10)

<table>
<thead>
<tr>
<th>Developer</th>
<th>Resolutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev14</td>
<td>37</td>
</tr>
<tr>
<td>Dev12</td>
<td>34</td>
</tr>
<tr>
<td>Dev31</td>
<td>24</td>
</tr>
<tr>
<td>Dev37</td>
<td>24</td>
</tr>
<tr>
<td>Dev36</td>
<td>21</td>
</tr>
<tr>
<td>Dev42</td>
<td>20</td>
</tr>
<tr>
<td>Dev18</td>
<td>14</td>
</tr>
<tr>
<td>Dev43</td>
<td>11</td>
</tr>
<tr>
<td>Dev24</td>
<td>11</td>
</tr>
<tr>
<td>Dev30</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.4: Files in the Python Project Containing Security Issues (Top 10)

<table>
<thead>
<tr>
<th>File</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects/unicodeobject.c</td>
<td>16</td>
</tr>
<tr>
<td>Modules/_ssl.c</td>
<td>15</td>
</tr>
<tr>
<td>Lib/ssl.py</td>
<td>13</td>
</tr>
<tr>
<td>Lib/sftpfile.py</td>
<td>11</td>
</tr>
<tr>
<td>Lib/platform.py</td>
<td>10</td>
</tr>
<tr>
<td>Lib/pty.py</td>
<td>10</td>
</tr>
<tr>
<td>Lib/smtpd.py</td>
<td>9</td>
</tr>
<tr>
<td>Python/parsermodule.c</td>
<td>9</td>
</tr>
<tr>
<td>Lib/posix.py</td>
<td>8</td>
</tr>
<tr>
<td>Python/myuprintf.c</td>
<td>8</td>
</tr>
</tbody>
</table>

Time to Fix

Time to Find
<table>
<thead>
<tr>
<th>Developer</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev00</td>
<td>14.29%</td>
</tr>
<tr>
<td>Dev01</td>
<td>4.35%</td>
</tr>
<tr>
<td>Dev02</td>
<td>3.23%</td>
</tr>
<tr>
<td>Dev03</td>
<td>2.78%</td>
</tr>
<tr>
<td>Dev04</td>
<td>2.53%</td>
</tr>
<tr>
<td>Dev05</td>
<td>2.08%</td>
</tr>
<tr>
<td>Dev06</td>
<td>1.59%</td>
</tr>
<tr>
<td>Dev07</td>
<td>1.59%</td>
</tr>
<tr>
<td>Dev08</td>
<td>1.19%</td>
</tr>
<tr>
<td>Dev09</td>
<td>1.15%</td>
</tr>
</tbody>
</table>

Table 4.1: Python Developer Vulnerability Injection Rate (Top 10)
Table 4.1: Python Developer Vulnerability Injection Rate (Top 10)

<table>
<thead>
<tr>
<th>Developer</th>
<th>Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev12</td>
<td>18</td>
</tr>
<tr>
<td>Dev44</td>
<td>10</td>
</tr>
<tr>
<td>Dev18</td>
<td>5</td>
</tr>
<tr>
<td>Dev45</td>
<td>3</td>
</tr>
<tr>
<td>Dev46</td>
<td>3</td>
</tr>
<tr>
<td>Dev47</td>
<td>3</td>
</tr>
<tr>
<td>Dev14</td>
<td>3</td>
</tr>
<tr>
<td>Dev24</td>
<td>3</td>
</tr>
<tr>
<td>Dev48</td>
<td>2</td>
</tr>
<tr>
<td>Dev30</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Python Developer Vulnerability Reports (Top 10)
### Table 4.3: Python Developer Vulnerability Resolutions (Top 10)

<table>
<thead>
<tr>
<th>File</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects/unicodeobject.c</td>
<td>16</td>
</tr>
<tr>
<td>Modules/_ssl.c</td>
<td>15</td>
</tr>
<tr>
<td>Lib/ssl.py</td>
<td>13</td>
</tr>
<tr>
<td>Lib/zipfile.py</td>
<td>11</td>
</tr>
<tr>
<td>Lib/platform.py</td>
<td>10</td>
</tr>
<tr>
<td>Lib/netrc.py</td>
<td>10</td>
</tr>
<tr>
<td>Lib/smtpd.py</td>
<td>9</td>
</tr>
<tr>
<td>Python/sysmodule.c</td>
<td>9</td>
</tr>
<tr>
<td>Lib/os.py</td>
<td>8</td>
</tr>
<tr>
<td>Python/mysnprintf.c</td>
<td>8</td>
</tr>
</tbody>
</table>

### Table 4.4: Files in the Python Project Containing Security Issues (Top 10)
<table>
<thead>
<tr>
<th>Developer</th>
<th>Resolutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev14</td>
<td>37</td>
</tr>
<tr>
<td>Dev12</td>
<td>34</td>
</tr>
<tr>
<td>Dev31</td>
<td>24</td>
</tr>
<tr>
<td>Dev37</td>
<td>24</td>
</tr>
<tr>
<td>Dev36</td>
<td>21</td>
</tr>
<tr>
<td>Dev42</td>
<td>20</td>
</tr>
<tr>
<td>Dev18</td>
<td>14</td>
</tr>
<tr>
<td>Dev43</td>
<td>11</td>
</tr>
<tr>
<td>Dev24</td>
<td>11</td>
</tr>
<tr>
<td>Dev30</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.3: Python Developer Vulnerability Resolutions (Top 10)
Time to Fix

- Under 31 Days
- 31-90 Days
- 91-120 Days
- 121-365 Days
- Longer than 1 Year
Time to Find

Pie chart showing time to find resources:
- Under 31 Days
- 31-365 Days
- 1-2 Years
- 2-5 Years
- Longer than 5 Years
Conclusions

- Novel approach for identifying vulnerability injection
- Good at finding new bugs that are similar to bugs that have already been fixed (and can sometimes reuse old fixes)
- Can still find bugs in projects with insufficient development metadata
- Almost entirely automated
Devin Cook
Yung Ryn Choe
Drew Hamilton
Auburn University
Sandia National Laboratories
CSET 2015

General Problem Statement

Finding Bugs in Source Code Using Commonly Available Development Metadata

Results

Conclusions

Background

Literature Review

Finding Bugs

Using Development Metadata

The Idea

Proposal

How it Works