Automated Tools for System and Application Security

John Mitchell
Outline

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states
• Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results
• Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding
• Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for related security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
Software bugs are serious problems

Thanks: Isil and Thomas Dillig
Facebook missed a single security check...

Man Finds Easy Hack to Delete Any Facebook Photo Album

*Facebook awards him a $12,500 "bug bounty" for his discovery*
App stores

Apps for whatever you’re up for.

Stay on top of the news. Stay on top of your finances. Or plan your dream vacation. No matter what you want to do with your iPhone, there’s probably an app to help you do it.

**Business**

iPhone is ready for work. Manage projects, track stocks, monitor finances, and more with these 9-to-5 apps.

View business apps in the App Store ›

**Education**

Keep up with your studies using intelligent education apps like King of Math and NatureTap.

View education apps in the App Store ›

**Entertainment**

Kick back and enjoy the show. Or find countless other ways to entertain yourself. These apps offer hours of viewing pleasure.

View entertainment apps in the App Store ›

**Family & Kids**

Turn every night into family night with interactive apps that are fun for the whole house.

View family and kids apps in the App Store ›

**Finance**

Create budgets, pay bills, and more with financial apps that take everything into account.

View finance apps in the App Store ›

**Food & Drink**


View food and drink apps in the App Store ›
How can you tell whether software you
– Develop
– Buy
is safe to install and run?
Cost of security or data privacy vulnerability?
Tools to help you
Manual testing only examines small subset of behaviors.
## Program Analyzers

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

**Diagram:**
- Code
- Spec
- Program Analyzer

**Table:**
<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>
Two options

• **Static analysis**
  – Automated methods to find errors or check their absence
    • Consider all possible inputs (in summary form)
    • Find bugs and vulnerabilities
    • Can prove absence of bugs, in some cases

• **Dynamic analysis**
  – Run instrumented code to find problems
    • Need to choose sample test input
    • Can find vulnerabilities but cannot prove their absence
Static Analysis

- Long research history
- Decades of commercial products
  - FindBugs, Fortify, Coverity, MS tools, ...
Dynamic analysis

• Instrument code for testing
  – Heap memory: Purify
  – Perl tainting (information flow)
  – Java race condition checking

• Black-box testing
  – Fuzzing and penetration testing
  – Black-box web application security analysis
Summary

• Program analyzers
  – Find problems in code before it is shipped to customers or before you install and run it

• Static analysis
  – Analyze code to determine behavior on all inputs

• Dynamic analysis
  – Choose some sample inputs and run code to see what happens
Outline

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states

• Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

• Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding

• Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for other security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
# Soundness, Completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>Equivalent Formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soundness</td>
<td>“Sound for reporting correctness” Analysis says no bugs → No bugs or equivalently There is a bug → Analysis finds a bug</td>
<td>$\neg B \rightarrow \neg A$</td>
</tr>
<tr>
<td>Completeness</td>
<td>“Complete for reporting correctness” No bugs → Analysis says no bugs</td>
<td>$\neg B \rightarrow \neg A$</td>
</tr>
<tr>
<td>Complete</td>
<td>Incomplete</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Sound</strong></td>
<td><strong>Incomplete</strong></td>
<td></td>
</tr>
<tr>
<td>Reports all errors</td>
<td>Reports all errors</td>
<td></td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
<td></td>
</tr>
<tr>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
<td></td>
</tr>
<tr>
<td>May not report all errors</td>
<td>May not report all errors</td>
<td></td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
<td></td>
</tr>
<tr>
<td><strong>Decidable</strong></td>
<td><strong>Decidable</strong></td>
<td></td>
</tr>
</tbody>
</table>
Sound Program Analyzer

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10,502</td>
<td>info leak</td>
<td>10,921</td>
</tr>
</tbody>
</table>

Analyze large code bases

Sound: may report many warnings

false alarm

false alarm
Sound
Over-approximation of Behaviors

False Alarm

approximation is too coarse...
...yields too many false alarms

Reported Error

Software
EXAMPLE

Program execution based on abstract states
Does this program ever crash?

```
entry

X ← 0

Is Y = 0 ?

<table>
<thead>
<tr>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ← X + 1</td>
<td>X ← X - 1</td>
</tr>
</tbody>
</table>

Is Y = 0 ?

<table>
<thead>
<tr>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is X &lt; 0 ?</td>
<td>exit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>crash</td>
<td></td>
</tr>
</tbody>
</table>
```
Does this program ever crash?

... program will never crash
Try analyzing without approximating...

non-termination!
... therefore, need to approximate
\[ d_{out} = f(d_{in}) \]
\[ d_{\text{out1}} = f_1(d_{\text{in1}}) \]
\[ d_{\text{out1}} = d_{\text{in2}} \]
\[ d_{\text{out2}} = f_2(d_{\text{in2}}) \]
What is the space of dataflow elements, \( \Delta \)?
What is the least upper bound operator, \( \sqcup \)?
Try analyzing with “signs” approximation...

```
X \leftarrow 0
```

```
Is Y = 0?
```

```
X = T
```

```
X = T
```

```
Is X < 0?
```

```
X = T
```

```
exit
```

```
crash
```

```
X = 0
```

```
X = 0
```

```
X = X + 1
```

```
X = X - 1
```

```
X = pos
```

```
X = neg
```

```
X = T
```

```
X = T
```

```
X = T
```

```
lost precision
```

```
terminates...
```

```
... but reports false alarm
```

```
... therefore, need more precision
```
Software

Sound Over-approximation of Behaviors

Reported Error

False Alarm

approximation is too coarse... yields too many false alarms
$X = T$

$X \neq \text{neg} \quad X = T \quad X \neq \text{pos}$

$X = \text{pos} \quad X = 0 \quad X = \text{neg}$

$X = \bot \quad X = \bot$

$Y = 0 \quad Y \neq 0$

true
false

refined signs lattice

Boolean formula lattice
Try analyzing with “path-sensitive signs” approximation...

entry

$X \leftarrow 0$

Is $Y = 0$?

$X \leftarrow X + 1$

$X \leftarrow X - 1$

Is $Y = 0$?

Is $X < 0$?

crash

... terminates...
... no false alarm
... soundly proved never crashes

true $X = 0$

$Y = 0$

$X = 0$

$X = pos$

$Y = 0$

$X = neg$

$Y = 0$

$X = var$

$Y = 0$

$X = pos$

$Y = 0$

no precision loss

refinement

no false alarm

soundly proved never crashes

terminates...
Summary of *sound* analysis

• Sound vs Complete
  – Cannot be sound and complete
  – Sound: can guarantee absence of bugs

• Sound analysis based on abstract states
  – Symbolically execute code using a description of all possible states at this program point
  – Better description: more precise, less efficient

• In practice
  – Use basic approach, possibly without soundness
    • E.g., do not run loops to termination
  – But keep the example in mind as good illustration
Outline

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states

Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

• Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding

• Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for other security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
Unsound Program Analyzer

analyze large code bases

false alarm

false alarm

Not sound: may miss some bugs

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mem leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer oflow</td>
<td>4,353,245</td>
</tr>
<tr>
<td>3</td>
<td>sql injection</td>
<td>23,212</td>
</tr>
<tr>
<td>4</td>
<td>stack oflow</td>
<td>86,923</td>
</tr>
<tr>
<td>5</td>
<td>dang ptr</td>
<td>8,491</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Bugs to Detect

Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code
- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Slide credit: Andy Chou
Example code with function def, calls

```c
#include <stdlib.h>
#include <stdio.h>

void say_hello(char * name, int size) {
    printf("Enter your name: ");
    fgets(name, size, stdin);
    printf("Hello %s.\n", name);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Error, must provide an input buffer size.\n");
        exit(-1);
    }
    int size = atoi(argv[1]);
    char * name = (char*)malloc(size);
    if (name) {
        say_hello(name, size);
        free(name);
    } else {
        printf("Failed to allocate %d bytes.\n", size);
    }
}
```
Reverse Topological Sort

Idea: analyze function before you analyze caller
Apply Library Models

Tool has built-in summaries of library function behavior
Bottom Up Analysis

Analyze function using known properties of functions it calls
Bottom Up Analysis

Analyze function using known properties of functions it calls
Bottom Up Analysis

Finish analysis by analyzing all functions in the program
Example: Check for missing optional args

• Prototype for open() syscall:

```c
int open(const char *path, int oflag, /* mode_t mode */...);
```

• Typical mistake:

```c
fd = open("file", O_CREAT);
```

• Result: file has random permissions

• Check: Look for oflags == O_CREAT without mode argument
Example: Chroot protocol checker

• **Goal:** confine process to a “jail” on the filesystem
  – chroot() changes filesystem root for a process

• **Problem**
  – chroot() itself does not change current working directory

```plaintext
chroot()
chdir("/")
open("../file", ...)
```

Error if open before chdir
• Race condition between time of check and use
• Not applicable to all programs
Tainting checkers

Tainted data accepted from source

Unvetted data taints other data transitively

Tainted data is used in an operator or function

Example Sinks: system() | printf() | malloc() | strcpy() | Sent to RDBMS | Included in HTML

Resultant Vulnerability: command injection | format string manip. | integer/ buffer overflow | buffer overflow | SQL injection | cross site scripting
Application to Security Bugs

• Stanford research project
  – Used modified compiler to find over 100 security holes in Linux and BSD
  – http://www.stanford.edu/~engler/

• Benefit
  – Capture recommended practices, known to experts, in tool available to all
Sanitize integers before use

Warn when unchecked integers from untrusted sources reach trusting sinks

Linux: 125 errors, 24 false; BSD: 12 errors, 4 false
Example security holes

• Remote exploit, no checks

/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
    msg = skb->data;
    ...
    memcpy(cmd.parm.setup.phone,
            msg->msg.connect_ind.addr.num,
            msg->msg.connect_ind.addr.len - 1);
Example security holes

• Missed lower-bound check:

/* 2.4.5/drivers/char/drm/i810_dma.c */

if(copy_from_user(&d, arg, sizeof(arg)))
    return -EFAULT;
if(d.idx > dma->buf_count)
    return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
User-pointer inference

• **Problem: which are the user pointers?**
  - Hard to determine by dataflow analysis
  - Easy to tell if kernel *believes* pointer is from user!

• **Belief inference**
  - “*p*” implies safe kernel pointer
  - “copyin(p)/copyout(p)” implies dangerous user ptr
  - Error: pointer p has both beliefs.

• **Implementation: 2 pass checker**
  inter-procedural: compute all tainted pointers
  local pass to check that they are not dereferenced
Environment Assumptions

- Should the return value of `malloc()` be checked?

```c
int *p = malloc(sizeof(int));
*p = 42;
```

<table>
<thead>
<tr>
<th>OS Kernel:</th>
<th>File server:</th>
<th>Web application:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash machine.</td>
<td>Pause filesystem.</td>
<td>200ms downtime</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spreadsheet:</th>
<th>Game:</th>
<th>IP Phone:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lose unsaved changes.</td>
<td>Annoy user.</td>
<td>Annoy user.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library:</th>
<th>Medical device:</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>malloc?!</td>
</tr>
</tbody>
</table>
• Assume the code is usually right

\[
\begin{align*}
\text{3/4 deref} & : \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \ast p = 42; \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \ast p = 42; \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \ast p = 42; \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \text{if(p) } \ast p = 42; \\
\end{align*}
\]

\[
\begin{align*}
\text{1/4 deref} & : \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \ast p = 42; \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \ast p = 42; \\
\text{int } \ast p = \text{malloc(sizeof(int))}; & \Rightarrow \text{int } \ast p = \text{malloc(sizeof(int))}; \\
& \text{if(p) } \ast p = 42; \\
\end{align*}
\]
### Results for BSD and Linux

- All bugs released to implementers; most serious fixed

<table>
<thead>
<tr>
<th>Violation</th>
<th>Linux Bug Fixed</th>
<th>Linux Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain control of system</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>Corrupt memory</td>
<td>43</td>
<td>17</td>
</tr>
<tr>
<td>Read arbitrary memory</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td>Denial of service</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>Minor</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>125</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Violation</th>
<th>BSD Bug Fixed</th>
<th>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of service</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minor</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
Outline

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states
• Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding
• Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for other security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
Survey of Web Vulnerability Tools

Local

Remote

>$100K total retail price
Test Vectors By Category

![Test Vector Percentage Distribution](image-url)
# Detecting Known Vulnerabilities

Vulnerabilities for previous versions of Drupal, phpBB2, and WordPress

<table>
<thead>
<tr>
<th>Category</th>
<th>Drupal 4.7.0</th>
<th></th>
<th>phpBB2 2.0.19</th>
<th></th>
<th>Wordpress 1.5strayhorn</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NVD</td>
<td>Scanner</td>
<td>NVD</td>
<td>Scanner</td>
<td>NVD</td>
<td>Scanner</td>
</tr>
<tr>
<td>XSS</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>SQLI</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>XCS</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Session</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>CSRF</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Info Leak</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Good: Info leak, Session  
Decent: XSS/SQLI  
Poor: XCS, CSRF (low vector count?)
Vulnerability Detection

Scanners Overall detection rate

- Malware: 0%
- Info leak: 31.2%
- Config: 32.5%
- Session: 26.5%
- SQL 2nd order: 0%
- SQL 1st order: 21.4%
- CSRF: 17.1%
- XCS: 14.4%
- XSS advance: 11.25%
- XSS type 2: 15%
- XSS type 1: 62%
Secure development
Experimental Study

• What factors most strongly influence the likely security of a new web site?
  – Developer training?
  – Developer team and commitment?
    • freelancer vs stock options in startup?
  – Programming language?
  – Library, development framework?

• How do we tell?
  – Can we use automated tools to reliably measure security in order to answer the question above?
Approach

• Develop a web application vulnerability metric
  – Combine reports of 4 leading commercial black box vulnerability scanners and

• Evaluate vulnerability metric
  – using historical benchmarks and our new sample of applications.

• Use vulnerability metric to examine the impact of three factors on web application security:
  – startup company or freelancers
  – developer security knowledge
  – Programming language framework
Data Collection and Analysis

• Evaluate 27 web applications
  – from 19 Silicon Valley startups and 8 outsourcing freelancers
  – using 5 programming languages.

• Correlate vulnerability rate with
  – Developed by startup company or freelancers
  – Extent of developer security knowledge (assessed by quiz)
  – Programming language used.
Comparison of scanner vulnerability detection
# Developer security self-assessment

## Quiz Categories and Question Summary

<table>
<thead>
<tr>
<th>Q</th>
<th>Category Covered</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SSL Configuration</td>
<td>Why CA PKI is needed</td>
</tr>
<tr>
<td>2</td>
<td>Cryptography</td>
<td>How to securely store passwords</td>
</tr>
<tr>
<td>3</td>
<td>Phishing</td>
<td>Why SiteKeys images are used</td>
</tr>
<tr>
<td>4</td>
<td>SQL Injection</td>
<td>Using prepared statements</td>
</tr>
<tr>
<td>5</td>
<td>SSL Configuration/XSS</td>
<td>Meaning of “secure” cookies</td>
</tr>
<tr>
<td>6</td>
<td>XSS</td>
<td>Meaning of “httponly” cookies</td>
</tr>
<tr>
<td>7</td>
<td>XSS/CSRF/Phishing</td>
<td>Risks of following emailed link</td>
</tr>
<tr>
<td>8</td>
<td>Injection</td>
<td>PHP local/remote file-include</td>
</tr>
<tr>
<td>9</td>
<td>XSS</td>
<td>Passive DOM-content intro. methods</td>
</tr>
<tr>
<td>10</td>
<td>Information Disclosure</td>
<td>Risks of auto-backup (“~”) files</td>
</tr>
<tr>
<td>11</td>
<td>XSS/Same-origin Policy</td>
<td>Consequence of error in Applet SOP</td>
</tr>
<tr>
<td>12</td>
<td>Phishing/Clickjacking</td>
<td>Risks of being iframed</td>
</tr>
</tbody>
</table>
Language usage in sample

<table>
<thead>
<tr>
<th>Language</th>
<th>Average Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>24,320</td>
</tr>
<tr>
<td>Java</td>
<td>14,630</td>
</tr>
<tr>
<td>PHP</td>
<td>17,020</td>
</tr>
<tr>
<td>Python</td>
<td>23,125</td>
</tr>
<tr>
<td>Ruby</td>
<td>7,660</td>
</tr>
</tbody>
</table>
Summary of developer study

• Security scanners are useful but not perfect
  – Tuned to current trends in web application development
  – Tool comparisons performed on single testbeds are not predictive in a statistically meaningful way
  – Combined output of several scanners is a reasonable comparative measure of code security, compared to other quantitative measures

• Based on scanner-based evaluation
  – Freelancers are more prone to introducing injection vulnerabilities than startup developers, in a statistically meaningful way
  – PHP applications have statistically significant higher rates of injection vulnerabilities than non-PHP applications; PHP applications tend not to use frameworks
  – Startup developers are more knowledgeable about cryptographic storage and same-origin policy compared to freelancers, again with statistical significance.
  – Low correlation between developer security knowledge and the vulnerability rates of their applications

Warning: don’t hire freelancers to build secure web site in PHP.
Outline

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states

• Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

• Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding

Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for other security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
STAMP Admission System

Static Analysis
More behaviors, fewer details

Dynamic Analysis
Fewer behaviors, more details

STAMP

Alex Aiken,
John Mitchell,
Saswat Anand,
Jason Franklin
Osbert Bastani,
Lazaro Clapp,
Patrick Mutchler,
Manolis Papadakis
Data Flow Analysis

• **Source-to-sink flows**
  - Sources: Location, Calendar, Contacts, Device ID etc.
  - Sinks: Internet, SMS, Disk, etc.
Applications of Data Flow Analysis

• **Malware/Greyware Analysis**
  - Data flow summaries enable enterprise-specific policies

• **API Misuse and Data Theft Detection**

• **Automatic Generation of App Privacy Policies**
  - Avoid liability, protect consumer privacy

• **Vulnerability Discovery**

Privacy Policy
This app collects your:
- Contacts
- Phone Number
- Address
Challenges

• Android is 3.4M+ lines of complex code
  o Uses reflection, callbacks, native code

• **Scalability**: Whole system analysis impractical

• **Soundness**: Avoid missing flows

• **Precision**: Minimize false positives
STAMP Approach

Too expensive!

• Model Android/Java
  ○ Sources and sinks
  ○ Data structures
  ○ Callbacks
  ○ 500+ models

• Whole-program analysis
  ○ Context sensitive
Building Models

• 30k+ methods in Java/Android API
  o 5 mins x 30k = 2500 hours

• Follow the permissions
  o 20 permissions for sensitive sources
    ▪ ACCESS_FINE_LOCATION (8 methods with source annotations)
    ▪ READ_PHONE_STATE - (9 methods)
  o 4 permissions for sensitive sinks
    ▪ INTERNET, SEND_SMS, etc.
Identifying Sensitive Data

- Returns device IMEI in String
- Requires permission GET_PHONE_STATE

@STAMP(
    SRC ="$GET_PHONE_STATE.deviceid",
    SINK ="@return"
)
Data We Track (Sources)

• Account data
• Audio
• Calendar
• Call log
• Camera
• Contacts
• Device Id
• Location
• Photos (Geotags)
• SD card data
• SMS

30+ types of sensitive data
Data Destinations (Sinks)

- Internet (socket)
- SMS
- Email
- System Logs
- Webview/Browser
- File System
- Broadcast Message

10+ types of exit points
Currently Detectable Flow Types

396 Flow Types

Unique Flow Types = Sources x Sink
Example Analysis

Contact Sync for Facebook (unofficial)

This app is compatible with your Sprint Samsung Nexus S 4G.

More from developer

Where Money Go? - DANUT CHERECHE<br>Free - 1 star (4)

Deschis - DANUT CHERECHE<br>Free - 1 star (6)

See more>

Users who viewed this also viewed

HaxSync - 4.x Facebook Sync - MATHIAS ROTH<br>$0.99

Friends Sync - WATTO STUDIOS

Visit Developer’s Website - Email Developer - Privacy Policy>

App Screenshots
## Contact Sync Permissions

<table>
<thead>
<tr>
<th>Category</th>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your Accounts</td>
<td>AUTHENTICATE_ACCOUNTS</td>
<td>Act as an account authenticator</td>
</tr>
<tr>
<td></td>
<td>MANAGE_ACCOUNTS</td>
<td>Manage accounts list</td>
</tr>
<tr>
<td></td>
<td>USE_CREDENTIALS</td>
<td>Use authentication credentials</td>
</tr>
<tr>
<td>Network Communication</td>
<td>INTERNET</td>
<td>Full Internet access</td>
</tr>
<tr>
<td></td>
<td>ACCESS_NETWORK_STATE</td>
<td>View network state</td>
</tr>
<tr>
<td>Your Personal Information</td>
<td>READ_CONTACTS</td>
<td>Read contact data</td>
</tr>
<tr>
<td></td>
<td>WRITE_CONTACTS</td>
<td>Write contact data</td>
</tr>
<tr>
<td>System Tools</td>
<td>WRITE_SETTINGS</td>
<td>Modify global system settings</td>
</tr>
<tr>
<td></td>
<td>WRITE_SYNC_SETTINGS</td>
<td>Write sync settings (e.g. Contact sync)</td>
</tr>
<tr>
<td></td>
<td>READ_SYNC_SETTINGS</td>
<td>Read whether sync is enabled</td>
</tr>
<tr>
<td></td>
<td>READ_SYNC_STATS</td>
<td>Read history of syncs</td>
</tr>
<tr>
<td>Your Accounts</td>
<td>GET_ACCOUNTS</td>
<td>Discover known accounts</td>
</tr>
<tr>
<td>Extra/Custom</td>
<td>WRITE_SECURE_SETTINGS</td>
<td>Modify secure system settings</td>
</tr>
</tbody>
</table>
Possible Flows from Permissions

Sources
- READ_CONTACTS
- READ_SYNC_SETTINGS
- READ_SYNC_STATS
- GET_ACCOUNTS
- INTERNET

Sinks
- INTERNET
- WRITE_SETTINGS
- WRITE_CONTACTS
- WRITE_SECURE_SETTINGS
- WRITE_SETTINGS
Expected Flows

**Sources**
- READ_CONTACTS
- READ_SYNC_SETTINGS
- READ_SYNC_STATS
- GET_ACCOUNTS
- INTERNET

**Sinks**
- INTERNET
- WRITE_SETTINGS
- WRITE_CONTACTS
- WRITE_SECURE_SETTINGS
- WRITE_SETTINGS
Example Study: Mobile Web Apps

• Goal
  Identify security concerns and vulnerabilities specific to mobile apps that access the web using an embedded browser

• Technical summary
  • WebView object renders web content
  • methods loadUrl, loadData, loadDataWithBaseUrl, postUrl
  • addJavascriptInterface(obj, name) allows JavaScript code in the web content to call Java object method name.foo()
Sample results

Analyze 998,286 free web apps from June 2014

<table>
<thead>
<tr>
<th>Mobile Web App Feature</th>
<th>% Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript Enabled</td>
<td>97</td>
</tr>
<tr>
<td>JavaScript Bridge</td>
<td>36</td>
</tr>
<tr>
<td>shouldOverrideUrlLoading</td>
<td>94</td>
</tr>
<tr>
<td>shouldInterceptRequest</td>
<td>47</td>
</tr>
<tr>
<td>onReceivedSslError</td>
<td>27</td>
</tr>
<tr>
<td>postUrl</td>
<td>2</td>
</tr>
<tr>
<td>Custom URL Patterns</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vuln</th>
<th>% Relevant</th>
<th>% Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsafe Navigation</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>Unsafe Retrieval</td>
<td>40</td>
<td>56</td>
</tr>
<tr>
<td>Unsafe SSL</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>Exposed POST</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Leaky URL</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>
Summary

• General discussion of code analysis tools
  – Goals and limitations of static, dynamic tools
  – Static analysis based on abstract states

• Security tools for traditional systems programming
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

• Web security analysis
  – Black-box security tools
  – Study based on these tools: security of coding

• Static analysis for Android malware
  – Determining whether app is malicious
  – Using tools for other security studies

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...