Program Analysis for Security

John Mitchell
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

[PopPhoto.com Feb 10]
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.

**LinkedIn: 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 >](#)

**Google: Education**

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

[View education apps in the App Store >](#)

**HBO Go: 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?
Two options

• Static analysis
  – Inspect code or run automated method to find errors or gain confidence about their absence

• Dynamic analysis
  – Run code, possibly under instrumented conditions, to see if there are likely problems
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>
Manual testing only examines small subset of behaviors
Static vs Dynamic Analysis

- **Static**
  - Can consider all possible inputs
  - Find bugs and vulnerabilities
  - Can prove absence of bugs, in some cases

- **Dynamic**
  - Need to choose sample test input
  - Can find bugs vulnerabilities
  - Cannot prove their absence
Cost of Fixing a Defect

Credit: Andy Chou, Coverity
Cost of security or data privacy vulnerability?
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
Static Analysis

• Long research history
• Decade of commercial products
  – FindBugs, Fortify, Coverity, MS tools, ...
Static Analysis: Outline

• General discussion of static analysis tools
  – Goals and limitations
  – Approach based on abstract states
• More about one specific approach
  – Property checkers from Engler et al., Coverity
  – Sample security checkers results
• Static analysis for of Android apps

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

• Bug finding
  – Identify code that the programmer wishes to modify or improve

• Correctness
  – Verify the absence of certain classes of errors
## Soundness, Completeness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Soundness</strong></td>
<td>“Sound for reporting correctness”</td>
</tr>
<tr>
<td></td>
<td>Analysis says no bugs $\rightarrow$ No bugs</td>
</tr>
<tr>
<td></td>
<td>or equivalently</td>
</tr>
<tr>
<td></td>
<td>There is a bug $\rightarrow$ Analysis finds a bug</td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>“Complete for reporting correctness”</td>
</tr>
<tr>
<td></td>
<td>No bugs $\rightarrow$ Analysis says no bugs</td>
</tr>
</tbody>
</table>

Recall: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$
<table>
<thead>
<tr>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td><strong>Unsound</strong></td>
</tr>
<tr>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td>Reports no false alarms</td>
<td>Reports no false alarms</td>
</tr>
<tr>
<td><strong>Decidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
</tbody>
</table>
Sound Program Analyzer

- Code
- Spec

Program Analyzer

- May report many warnings
- May emit false alarms
- Analyze large code bases

- 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>

- False alarms:
  - Buffer oflow
  - Stack oflow

- Sound: may report many warnings
Software

**Over-approximation of Behaviors**

- Sound: Reported Error
- False Alarm

Approximation is too coarse...
...yields too many false alarms
Outline

• General discussion of tools
  – Goals and limitations
  Approach based on abstract states
• More about one specific approach
  – Property checkers from Engler et al., Coverity
  – Sample security-related results
• Static analysis for Android malware
  – ...

Slides from: S. Bugrahe, A. Chou, I&T Dillig, D. Engler, J. Franklin, A. Aiken, ...
Does this program ever crash?

```
entry

X ← 0

Is Y = 0 ?

X ← X + 1  X ← X - 1

Is Y = 0 ?

Is X < 0 ?

exit

crash
```
Does this program ever crash?

```
Does this program ever crash?

event

entry

X ← 0

Is Y = 0 ?

Is Y = 0 ?

X ← X + 1

X ← X - 1

Is X < 0 ?

Is X < 0 ?

crash

exit

infeasible path!
... program will never crash
```
Try analyzing without approximating...

```
X <- 0
Is Y = 0?

yes
X <- X + 1
X <- X - 1
Is Y = 0?

yes
Is X < 0?

yes

... therefore, need to approximate

X = 3
```
dataflow elements

transfer function

dataflow equation

\[ d_{\text{out}} = f(d_{\text{in}}) \]
\[ X = 0 \]

\[ X \leftarrow X + 1 \]

\[ X = 1 \]

\[ \text{Is } Y = 0 ? \]

\[ X = 1 \]

\[ X = 1 \]

\[ X = 1 \]

\[ d_{in1} \]

\[ d_{out1} \]

\[ d_{in2} \]

\[ d_{out2} \]

\[ f1 \]

\[ d_{out1} = f_1(d_{in1}) \]

\[ d_{out1} = d_{in2} \]

\[ d_{out2} = f_2(d_{in2}) \]
What is the space of dataflow elements, \( \Delta \)?
What is the least upper bound operator, \( \sqcup \)?

Least upper bound operator
Example: union of possible values

\[
\begin{align*}
d_{out1} &= f_1(d_{in1}) \\
d_{out2} &= f_2(d_{in2}) \\
d_{join} &= d_{out1} \sqcup d_{out2} \\
d_{in3} &= \text{d}_{in3} \\
d_{out3} &= f_3(d_{in3})
\end{align*}
\]
Try analyzing with “signs” approximation...

X \leftarrow X + 1 \quad \text{Is } X < 0 \quad \text{?}

\begin{align*}
X &= 0 \\
X &= pos \\
X &= T \\
\end{align*}

X \leftarrow X - 1 \\
\begin{align*}
X &= 0 \\
X &= neg \\
X &= T \\
\end{align*}

\begin{align*}
\text{is } Y &= 0 \quad \text{yes} \\
\text{is } Y &= 0 \quad \text{no}
\end{align*}

\begin{align*}
\text{is } X &= 0 \quad \text{yes} \\
\text{is } X &= 0 \quad \text{no}
\end{align*}

\begin{align*}
\text{crash} \\
\text{exit}
\end{align*}

... but reports false alarm
... therefore, need more precision
\[ X = T \]
\[ X \neq \neg \]
\[ X = \text{pos} \]
\[ X = 0 \]
\[ X = \text{neg} \]
\[ X = \bot \]
\[ X = \bot \]
\[ X = \text{false} \]

\[ Y = 0 \]
\[ Y \neq 0 \]
\[ \text{true} \]
\[ \text{false} \]
Try analyzing with “path-sensitive signs” approximation...

entry

\[ X \leftarrow 0 \]

Is \( Y = 0 \)?

- true: \( X = 0 \)
- \( Y = 0 \): \( X = 0 \)
- \( Y = 0 \): \( X = \text{pos} \)
- \( Y \neq 0 \):
  - \( X = \text{pos} \)
  - \( X = \text{neg} \)

\[ X \leftarrow X + 1 \]
\[ X \leftarrow X - 1 \]

Is \( Y = 0 \)?

- yes: \( X = \text{pos} \)
- no: \( X = \text{neg} \)

Is \( X < 0 \)?

- yes: \( X = \text{pos} \)
- no: \( X = \text{neg} \)

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

exit

crash
Outline

• General discussion of tools
  – Goals and limitations
  – Approach based on abstract states

★ More about one specific approach
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

• Static analysis for Android malware
  – ...

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

Code → Program Analyzer → Spec

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>
</tbody>
</table>

analyze large code bases

false alarm

false alarm

may emit false alarms
Demo

• Coverity video: [http://youtu.be/_Vt4niZfNeA](http://youtu.be/_Vt4niZfNeA)

• Observations
  – Code analysis integrated into development workflow
  – Program context important: analysis involves sequence of function calls, surrounding statements
  – This is a sales video: no discussion of false alarms
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
- Unused return codes
- Use of invalid iterators

Slide credit: Andy Chou
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

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

Error if open before chdir
TOCTOU

- 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
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
#define SIZE 8
void set_a_b(char * a, char * b) {
    char * buf[SIZE];
    if (a) {
        b = new char[5];
    } else {
        if (a && b) {
            buf[SIZE] = a;
            return;
        } else {
            delete [] b;
        }
        *b = 'x';
    }
    *a = *b;
}
char * buf[8];

if (a)
{
    b = new char[5];
    if (a && b)
    {
        buf[8] = a;
        delete[] b;
        *b = 'x';
        *a = *b;
    }
    END
}

Control Flow Graph

Represent logical structure of code in graph form
Path Traversal

char * buf[8];

if (a)
  b = new char [5];
if (a && b)
  buf[8] = a;
delete [] b;

*a = *b;

END

Conceptually: Analyze each path through control graph separately

Actually: Perform some checking computation once per node; combine paths at merge nodes
Apply Checking

Null pointers Use after free Array overrun

See how three checkers are run for this path

Checker
- Defined by a state diagram, with state transitions and error states

Run Checker
- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached
Apply Checking

Null pointers Use after free Array overrun

`char * buf[8];`

`if (a)`

`!a`

`if (a && b)`

`!(a && b)`

`delete [] b;`

`*b = ‘x’;`

`*a = *b;`

END

“buf is 8 bytes”
Apply Checking

Null pointers Use after free Array overrun

char * buf[8];
if (a)
!a
if (a && b)
!(a && b)
delete [] b;
*b = 'x';
*a = *b;
END

“buf is 8 bytes”

“a is null”
Apply Checking

Null pointers Use after free Array overrun

char * buf[8];
if (a)
!a
if (a && b)
!(a && b)
delete [] b;
*b = 'x';
*a = *b;
END

“buf is 8 bytes”

“a is null”

Already knew a was null
char * buf[8];

if (a)

!a

if (a && b)

!(a && b)

delete [] b;

*b = 'x';

*a = *b;

END

Null pointers Use after freeArray overrun

“buf is 8 bytes”

“a is null”

“b is deleted”
Apply Checking

**Null pointers Use after free Array overrun**

```
char * buf[8];

if (a)
!a
if (a && b)
!(a && b)
delete [] b;
*b = 'x';
*a = *b;
END
```

- "buf is 8 bytes"
- "a is null"
- "b is deleted"
- "b dereferenced!"
Apply Checking

Null pointers Use after free Array overrun

char * buf[8];
if (a)
!a
if (a && b)
!(a && b)
delete [] b;
*b = 'x';
*a = *b;
END

“buf is 8 bytes”
“a is null”
“b is deleted”
“b dereferenced!”
No more errors reported for b
False Positives

• What is a bug? Something the user will fix.

• Many sources of false positives
  – False paths
  – Idioms
  – Execution environment assumptions
  – Killpaths
  – Conditional compilation
  – “third party code”
  – Analysis imprecision
  – …
char * buf[8];

if (a)
    b = new char [5];

if (a && b)
    buf[8] = a;

delete [] b;

*b = 'x';

*a = *b;

END
False Path Pruning

char * buf[8];

if (a)

!a

if (a && b)

buf[8] = a;

END
False Path Pruning

```c
char * buf[8];
if (a)
!a
if (a && b)
a && b
buf[8] = a;
```

“a in [0,0]”
“a == 0 is true”
```c
char * buf[8];

if (a)
    if (a && b)
        buf[8] = a;

END
```

**False Path Pruning**

```
"a in [0,0]"
"a == 0 is true"
"a != 0"
```

```
"a in [0,0]"
"a == 0 is true"
"a != 0"
```
**False Path Pruning**

```
char * buf[8];
if (a)
    !a
if (a && b)
    a && b
buf[8] = a;
```

Impossible

- "a in [0,0]"
- "a == 0 is true"
- "a != 0"
- "a != 0 is true"
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>
Statistical Analysis

• Assume the code is usually right

\[
\begin{align*}
\text{int } *p &= \text{malloc(sizeof(int))}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
\text{if}(p) *p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
\text{if}(p) *p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
\text{if}(p) *p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc(sizeof(int))}; \\
\text{if}(p) *p &= 42; \\
\end{align*}
\]
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/](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 */
isdnctlr 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:

```c
/* 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
Results for BSD and Linux

- All bugs released to implementers; most serious fixed

<table>
<thead>
<tr>
<th>Violation</th>
<th>Linux Bug</th>
<th>Linux Fixed</th>
<th>BSD Bug</th>
<th>BSD Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain control of system</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Corrupt memory</td>
<td>43</td>
<td>17</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Read arbitrary memory</td>
<td>19</td>
<td>14</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Denial of service</td>
<td>17</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Minor</td>
<td>28</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>125</td>
<td>52</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>
Outline

• General discussion of tools
  – Goals and limitations
  – Approach based on abstract states

• More about one specific approach
  – Property checkers from Engler et al., Coverity
  – Sample security-related results

★ Static analysis for Android malware
  – ...

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

Dynamic Analysis
Fewer behaviors, more details

Static Analysis
More behaviors, fewer details

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.

---

### Diagram

- **getLoc()** → **Source: Location** → **sendSMS()** → **Sink: SMS**
- **sendInet()** → **Sink: Internet**

- **Location** → **SMS** → **Location** → **Internet**
Applications of Data Flow Analysis

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

• API Misuse and Data Theft Detection

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

• Vulnerability Discovery
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

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

- Whole-program analysis
  - Context sensitive

Too expensive!
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.
android.Telephony.TelephonyManager: String getDeviceId()

• 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)
## 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>
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

• Static vs dynamic analyzers
• General properties of static analyzers
  – Fundamental limitations
  – Basic method based on abstract states
• More details on one specific method
  – Property checkers from Engler et al., Coverity
  – Sample security-related results
• Static analysis for Android malware
  – STAMP method, sample studies

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