Program Analysis for Security

Original slides created by Prof. John Mitchell
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?
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

Code

Spec

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>
So.ware

Behaviors

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>Soundness</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>Completeness</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></th>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Reports all errors</td>
<td>Reports all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td><strong>Undecidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
<tr>
<td><strong>Unsound</strong></td>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td></td>
<td>Reports no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td><strong>Decidable</strong></td>
<td><strong>Decidable</strong></td>
</tr>
</tbody>
</table>
Sound Program Analyzer

Sound: may report many warnings

May emit false alarms

False alarm

Analyze large code bases
Software

Behavior

Sound Over-approximation of Behaviors

False Alarm

approximation is too coarse...

...yields too many false alarms

Redacted Error

Redacted

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 ?

- yes: X ← X + 1
- no: X ← X - 1

Is Y = 0 ?

- yes: Is X < 0 ?
- no: exit

- yes: crash
- no: Is X < 0 ?
```
Does this program ever crash?

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

non-termination!
... therefore, need to approximate
$X = 0$

$X \leftarrow X + 1$

$X = 1$

$d_{in}$

$d_{out} = f(d_{in})$

Transfer function

dataflow equation

Dataflow elements
\[ X = 0 \]

\[ X \leftarrow X + 1 \]

\[ X = 1 \]

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

\[ X = 1 \]

\[ X = 1 \]

\[ d_{\text{in1}} \]

\[ d_{\text{out1}} \]

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

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

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

d_{out1} = f_1(d_{in1})

d_{out2} = f_2(d_{in2})

d_{join} = d_{out1} \sqcup d_{out2}

d_{join} = d_{in3}

d_{out3} = f_3(d_{in3})

least upper bound operator
Example: union of possible values
Try analyzing with “signs” approximation...

entry

\[ X \leftarrow 0 \]

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

\[ X = 0 \]

\[ X = 0 \]

\[ X = \text{pos} \]

\[ X = \overline{\text{T}} \]

\[ X = \text{T} \]

\[ X = \text{neg} \]

\[ X = \text{T} \]

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

\[ X \leftarrow X + 1 \]

\[ X \leftarrow X - 1 \]

\[ \text{Is } X < 0? \]

\[ \text{exit} \]

\[ \text{crash} \]

lost precision

terminates...
... but reports false alarm
... therefore, need more precision
\( X = T \)

\( X = \neg \neg X \)

\( X = \neg pos \)

\( X = \neg neg \)

\( X = 0 \)

\( X = \bot \)

\( Y = 0 \)

\( Y = \neg 0 \)

\( true \)

\( false \)

Refinement Lattice

Boolean Formula Lattice
Try analyzing with “path-sensitive signs” approximation...

entry

X ← 0

Is Y = 0?

Y=0, X=0

Y=0, X=pos

Y=0, X=neg

X ← X + 1

X ← X - 1

Is Y = 0?

Is X < 0?

X = 0

X = neg

X = pos

X = pos

X = neg

crash

exit

no precision loss

refinement

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

Not sound: may miss some bugs

May emit false alarms

Analyze large code bases

False alarm

False alarm

<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>
Build Info

Source Code

Virtual Build Environment

- /prevent
  - Quality
  - Security
  - Concurrency
- /extend
  - Custom Checks

Analysis Engine

- Interprocedural
- Dataflow Analysis
- Statistical Analysis
- False Path Pruning
- 100% of All Paths
- Incremental Analysis

Defect Manager

- Report Dashboard
- Developer Dashboard
- IDE Support
- Open Standard Interfaces
Demo

• Coverity video: 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
- Unhandled 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
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
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];
    *b = 'x';
    *a = *b;
}

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

!(a && b)

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

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

char * buf[8];

if (a)
!a
if (a && b)
!(a && b)
delete [] b;
*b = ‘x’;
*a = *b;
END

Null pointers Use after free Array overrun

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

Apply Checking

Null pointers Use after free Array overrun

“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
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”
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;
```

“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
char * buf[8];

if (a)

if (a && b)

buf[8] = a;

END
False Path Pruning

```c
char * buf[8];

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

END
```

 integer range disequality branch

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

"a && b"
False Path Pruning

char * buf[8];

if (a)

!a

if (a && b)

a && b

buf[8] = a;

END

“a in [0,0]”

“a == 0 is true”

“a != 0”

“a != 0”
False Path Pruning

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

Impossible

- “a in [0,0]”
- “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

3/4 deref

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

1/4 deref

\[
\begin{align*}
\text{int } *p &= \text{malloc}() \text{sizeof(int)}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc}() \text{sizeof(int)}; \\
*p &= 42; \\
\text{int } *p &= \text{malloc}() \text{sizeof(int)}; \\
\text{if(p)} *p &= 42;
\end{align*}
\]
Example security holes

• Remote exploit, no checks

/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdnn_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:

```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);
```
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, ...