Automated Tools for System and Application Security

John Mitchell
Outline

› Introduction: static vs dynamic analysis
› Static analysis
  ▪ Program execution using state descriptions
  ▪ Security examples: static analysis to find vulnerabilities
› Dynamic vulnerability finding
  ▪ Black-box testing: principles and web examples
  ▪ Fuzzing: principles and examples
Users of these methods and tools

- Engineers
- Criminals
- Security Researchers
- Pen Testers
- Governments
- Hacktivists
- Academics

Remember this:
If you develop code, you should test it using the same methods that attackers will use against you
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
Engineering challenges

People care about features, not security (until something goes wrong)
Engineers typically only see a small piece of the puzzle
“OMG PDF WTF” (Julia Wolf, 2010)

- How many lines of code in Linux 2.6.32?
  › 10 million
- How many lines in Windows NT 4?
  › 11-12 million
- How many in Adobe Acrobat?
  › 13 million
Summary

› Program bugs are ubiquitous
› Security vulnerabilities are common too
  ▪ Not all bugs lead to exploits
  ▪ Exploits are serious and occur in many systems
› Program bugs can be found through systematic methods
› Better to find bugs and vulnerabilities early
  ▪ Before attackers find and exploit them
› Use tools and systematic methods in development, Q/A, later
Outline

Introduction: static vs dynamic analysis

- Static analysis
  - Program execution using state descriptions
  - Security examples: static analysis to find vulnerabilities

- Dynamic vulnerability finding
  - Black-box testing: principles and web examples
  - Fuzzing: principles and examples
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>
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
Software

Behaviors

1 → 2 → 4
1 → 3 → 4
1 → 2 → 4 → 1 → 2 → 4
1 → 2 → 4 → 1 → 3 → 4
1 → 2 → 3 → 1 → 2 → 4 → 1 → 3 → 4
1 → 2 → 4 → 1 → 2 → 3 → 1 → 3 → 4
1 → 2 → 3 → 1 → 2 → 3 → 1 → 3 → 4
1 → 2 → 4 → 1 → 2 → 4 → 1 → 3 → 4

...
Dynamic testing examines subset of behaviors.
Static testing uses abstraction to consider all behaviors.
Static Analysis

Long history of academic research

Decades of commercial products
- FindBugs, Fortify, Coverity, MS tools, ...

Commonly used in current practice
- Teams use development process and unit tests
- Static, dynamic tools find certain classes of bugs in dev, Q/A
- After release, hackers and support teams look for vulns
Dynamic analysis

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

**Black-box testing**
- Black-box web application security analysis
- Fuzzing and penetration testing
Comparison 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
- Sample inputs can be based on code analysis

In practice, static and dynamic analysis are often combined
Outline

› Introduction: static vs dynamic analysis

Static analysis
- Program execution using state descriptions
- Security examples: static analysis to find vulnerabilities

› Dynamic vulnerability finding
- Black-box testing: principles and web examples
- Fuzzing: principles and examples
“Sound” Program Analyzer

Verify absence of vulnerabilities

Program Analyzer

Code

Spec

<table>
<thead>
<tr>
<th>Report</th>
<th>Type</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>memory leak</td>
<td>324</td>
</tr>
<tr>
<td>2</td>
<td>buffer overflow</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 overflow</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>

Sound: may report many warnings

Analyze large code bases

false alarm

false alarm
## 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 → No bugs</td>
</tr>
<tr>
<td></td>
<td><em>or equivalently</em></td>
</tr>
<tr>
<td></td>
<td>There is a bug → Analysis finds a bug</td>
</tr>
<tr>
<td>Completeness</td>
<td>“Complete for reporting correctness”</td>
</tr>
<tr>
<td></td>
<td>No bugs → Analysis says no bugs</td>
</tr>
</tbody>
</table>

Fact from logic: \( A \rightarrow B \) 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>
Software

Behaviors

Over-approximation of Behaviors

False Alarm

Sound

approximation is too coarse...

Error

...yields too many false alarms

Reported

Modules
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>X ← X + 1</td>
<td>X ← X - 1</td>
</tr>
</tbody>
</table>

Is X < 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>

exit

crash
```
Does this program ever crash?

entry

\[ X \leftarrow 0 \]

Is \( Y = 0 \) ?

yes

\[ X \leftarrow X + 1 \]

no

\[ X \leftarrow X - 1 \]

Is \( Y = 0 \) ?

yes

no

Is \( X < 0 \) ?

yes

\[ \text{crash} \]

no

exit

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

entry

X ← 0

Is Y = 0 ?

X = 0

Is Y = 0 ?

X = 0

X ← X + 1

X = 2

yes

X ← X - 1

X = 3

no

Is Y = 0 ?

X = 3

Is Y = 0 ?

X = 3

Is X < 0 ?

X = 1

exit

no

yes

X = 3

crash

non-termination!

... therefore, need to approximate
\[ X = 0 \]

\[ X \leftarrow X + 1 \]

\[ X = 1 \]

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

dataflow elements

dataflow equation

transfer function
Is $Y = 0$?

$$X \leftarrow X + 1$$

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

$$d_{out1} = d_{in2}$$

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

Example: union of possible values

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

...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
\[
\begin{align*}
X &= T \\
X &\neq \text{neg} \\
X &\neq \text{pos} \\
X &= \text{pos} \\
X &= 0 \\
X &= \text{neg} \\
X &= \bot \\
\end{align*}
\]

\[
\begin{align*}
Y &= 0 \\
Y &\neq 0 \\
\end{align*}
\]

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

Is \( Y = 0 ? \)

\[ X \leftarrow 0 \]

Is \( Y = 0 ? \)

\[ X \leftarrow X + 1 \]

\[ X \leftarrow X - 1 \]

Is \( X < 0 ? \)

exit

true \[ X = 0 \]

1. \[ Y = 0 \] \[ X = 0 \]
2. \[ Y = 0 \] \[ X = \text{pos} \]
3. \[ Y = 0 \] \[ X = \text{neg} \]

no precision loss
refinement

terminates...
... no false alarm
... soundly proved never crashes
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 each program point
- Better descriptions $\Rightarrow$ more precise analysis, takes longer to analyze

In practice
- Use the basic approach design for soundness, but possibly without soundness
  - E.g., do not run loops to termination
- We discussed sound analysis using abstract states to understand the principle
Outline

› Introduction: static vs dynamic analysis
› Static analysis
   ▪ Program execution using state descriptions
   ▪ Security examples: static analysis to find vulnerabilities
› Dynamic vulnerability finding
   ▪ Black-box testing: principles and web examples
   ▪ Fuzzing: principles and examples
Program Analyzers in Practice

<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>

Not sound: may miss some bugs

analyze large code bases

false alarm

false alarm
Bugs to Detect

- 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:

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

Typical mistake:

```
f = 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() changes filesystem root for a process
chdir("/")
open("../file",...)
```

Error if open before chdir
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/

Commercialized and extended

Benefit

- Capture recommended expert practices in tool for everyone
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:

```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 make sure they are not dereferenced
Should the return value of malloc() be checked?

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

OS Kernel: Crash machine.
File server: Pause filesystem.
Web application: 200ms downtime

Spreadsheet: Lose unsaved changes.
Game: Annoy user.
IP Phone: Annoy user.

Library: ?
Medical device: malloc?!
Statistical Analysis

Assume the code is usually right

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

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

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

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

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

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

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

\[
\begin{align*}
\text{int } *p &= \text{malloc}(\text{sizeof}(\text{int})); \\
* p &= 42; \\
\end{align*}
\]
# 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>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain control of system</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>Corrupt memory</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>Read arbitrary memory</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>Denial of service</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>Minor</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>125</td>
<td>12</td>
</tr>
</tbody>
</table>
Summary of security analysis

Static tools can detect many classes of errors

- Protocol errors: libraries require function calls in specific sequences, e.g. open(file)
- Tainting errors: untrusted values must be checked before specific kinds of use

Automated tools used to find vulnerabilities in widely used systems

- Operating systems, device drivers, user applications, ...

Some simplifications used to reduce probability of false alarms

- Look for programmer inconsistencies – more useful to report if programmer confused
- Often more useful to report 10 significant bugs than a list of 1000 containing 11
Outline

› Introduction: static vs dynamic analysis
› Static analysis
   ▪ Program execution using state descriptions
   ▪ Security examples: static analysis to find vulnerabilities

Dynamic vulnerability finding
   ▪ Black-box testing: principles and web examples
   ▪ Fuzzing: principles and examples
Dynamic analysis

Basics
- Run code using many inputs
- Look for error or evidence of vulnerability

Successful approaches require
- Some way to choose inputs that will cause problems
- Methods to tell what happened (instrumentation, error trace)
Dynamic analysis methods

**Instrument code and test using programmer-generated input**
- Heap memory: Purify
- Perl tainting (information flow)
- Java race condition checking

**Testing using automated input generation**
- Black-box web application security analysis
- Fuzzing and penetration testing
Survey of black-box web security tools

Local
- IBM
- Acunetix
- Rapid7
- N-Stalker
- Cenzic

Remote
- McAfee Secure
- Qualys
- [Tested: 02-FEB]
Example scanner UI
Test Vectors By Category

Test Vectors Percentage Distribution

- Info leaks: 50% (50 test vectors)
- Configuration: 5% (5 test vectors)
- CSRF: 1% (1 test vector)
- Session: 10% (10 test vectors)
- XCS: 10% (10 test vectors)
- SQLI: 30% (30 test vectors)
- XSS: 20% (20 test vectors)
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>phpBB2 2.0.19</th>
<th>Wordpress 1.5strayhorn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NVD</td>
<td>Scanner</td>
<td>NVD</td>
</tr>
<tr>
<td>XSS</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>SQLI</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>XCS</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Session</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>CSRF</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Info Leak</td>
<td>4</td>
<td>3</td>
<td>1</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%
Summary of dynamic and black-box analysis

› Dynamic analysis tools can help detect many classes of errors
  ▪ Instrument code to detect error at program point where it occurs
  ▪ Use unit testing, other methods, based on developer team process

› Black-box tools can find important vulnerabilities
  ▪ Typically find common vulnerabilities that have been found elsewhere
  ▪ Typically poor at finding new or complex vulnerabilities
  ▪ Save time by finding common mistakes but do not guarantee security
Outline

› Introduction: static vs dynamic analysis
› Static analysis
  ▪ Program execution using state descriptions
  ▪ Security examples: static analysis to find vulnerabilities
› Dynamic vulnerability finding
  ▪ Black-box testing: principles and web examples
  ▶ Fuzzing: principles and examples
Fuzzing

› A form of dynamic vulnerability testing
› Generate “random” input based on any partial information
  ▪ Try many anomalous test cases
› Monitor execution to see if this produces crash or error
  ▪ Some kinds of errors can be used to find an exploit
Example

Standard HTTP GET request
- GET /index.html HTTP/1.1

Anomalous requests
- GEEEE...EET /index.html HTTP/1.1
- GET ///////////index.html HTTP/1.1
- GET %n%n%n%n%n%n.html HTTP/1.1
- GET /AAAAAAAAAAAAA.html HTTP/1.1
- GET /index.html HTTTTTTTTTTTTTTTTTTP/1.1
- GET /index.html HTTP/1.1.1.1.1.1.1.1
- etc...
Basic approaches

**Mutation-based fuzzing**
- Add anomalies to existing good inputs (e.g., test suite)

**Generation-based fuzzing**
- Generate inputs from specification of format, protocol, etc

**Responsive fuzzing**
- Leverage program instrumentation, code analysis
- Use response of program to build input set
Mutation-based fuzzing

Basic idea
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics

Advantages
- Little or no knowledge of the structure of the inputs is assumed
- Requires little to no set up time

Disadvantages
- Dependent on the inputs being modified
- May fail for protocols with checksums, challenge-response, etc.

Examples fuzzing tools:
- Taof, GPF, ProxyFuzz, etc.
Example: find bugs in pdf viewer

PDF Viewer

Crash viewer and isolate cause

Modify conventional pdf files

Will see example of clever pdf vulnerability finding later; not likely to succeed by chance.
Generation-based fuzzing

Basic idea
- Test cases generated from description of format: RFC, spec, etc.
- Anomalies are added to each possible spot in the inputs

Advantages
- Knowledge of protocol may give better results than random fuzzing

Disadvantages
- Can take significant time to set up

Example tools:
- SPIKE, Sulley, Mu-4000, Codenomicon
Example: network protocol bugs

SYN

SYN/ACK

ACK

TCP Header

IP Header

Source Port

Dest port

SEQ Number

ACK Number

Other stuff
Example: Specification of Zip file

```xml
<!-- A. Local file header -->
<Block name="LocalFileHeader">
  <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
  <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
  ...
  [truncated for space]
  ...
  <Number name="lfh_CompSize" size="32" endian="little" signed="false">
    <Relation type="size" of="lfh_CompData"/>
  </Number>
</Block>
```
Evolutionary fuzzing

Basic idea
- Generate inputs based on the structure, response of the program

Advantages
- Exploits detailed knowledge of program
- Prioritizes inputs that may lead to vulnerability

Disadvantages
- Requires more effort, may not prove successful

Examples:
- Autodafe
  - Prioritizes test cases based on which inputs have reached dangerous API functions
- EFS
  - Generates test cases based on code coverage metrics
AFL Fuzzing Tool (used in your lab)

Two basic components

- Compile-time instrumentation of code you are testing
- Genetic algorithms to automatically generate test cases that reach new specific internal states in the targeted binaries
AFL binary instrumentation

Two options:
1. **Wrapper for GCC/CLang compilers**
   - Injects assembly code during compilation
   - Fuzzer uses this code to trace execution paths
   - Fuzzer tests whether new inputs can reach new execution paths
2. **QEMU “user emulation” mode**
   - Instruments execution of black-box, closed-source binaries
   - Allows fuzzer to stress-test targets by same method as above, without recompilation
AFL example test

cd ~/binutils-2.25
CC=afl-gcc ./configure
make
echo core > /proc/sys/kernel/core_pattern
        -- write core to file
cd ~/binutils-2.25
mkdir afl_in afl_out
cp /bin/ps afl_in/
cd ~/binutils-2.25
afl-fuzz -i afl_in -o afl_out ./binutils/readelf -a @@

https://www.evilsocket.net/2015/04/30/fuzzing-with-afl-fuzz-a-practical-example-afl-vs-binutils/
<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time: 0 days, 0 hrs, 8 min, 24 sec</td>
<td>cycles done: 0</td>
</tr>
<tr>
<td>last new path: 0 days, 0 hrs, 1 min, 59 sec</td>
<td>total paths: 812</td>
</tr>
<tr>
<td>last uniq crash: 0 days, 0 hrs, 3 min, 17 sec</td>
<td>uniq crashes: 8</td>
</tr>
<tr>
<td>last uniq hang: 0 days, 0 hrs, 3 min, 23 sec</td>
<td>uniq hangs: 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing: 0 (0.00%)</td>
<td>map density: 3158 (4.82%)</td>
</tr>
<tr>
<td>paths timed out: 0 (0.00%)</td>
<td>count coverage: 2.56 bits/tuple</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>findings in depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying: arith 8/8</td>
<td>favored paths: 1 (0.12%)</td>
</tr>
<tr>
<td>stage execs: 295k/326k 90.31%</td>
<td>new edges on: 318 (39.16%)</td>
</tr>
<tr>
<td>total execs: 552k</td>
<td>total crashes: 63 (8 unique)</td>
</tr>
<tr>
<td>exec speed: 1114/sec</td>
<td>total hangs: 191 (10 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fuzzing strategy yields</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit flips: 447/75.5k, 59/75.5k, 59/75.5k</td>
<td>levels: 2</td>
</tr>
<tr>
<td>byte flips: 7/9436, 0/5858, 6/5950</td>
<td>pending: 812</td>
</tr>
<tr>
<td>arithmetics: 0/0, 0/0, 0/0</td>
<td>pend fav: 1</td>
</tr>
<tr>
<td>known ints: 0/0, 0/0, 0/0</td>
<td>own finds: 811</td>
</tr>
<tr>
<td>dictionary: 0/0, 0/0, 0/0</td>
<td>imported: n/a</td>
</tr>
<tr>
<td>havoc: 0/0, 0/0</td>
<td>variable: 0</td>
</tr>
<tr>
<td>trim: 0.00%/1166, 38.39%</td>
<td></td>
</tr>
</tbody>
</table>

cpu: 15%
Summary

Fuzzing: dynamic vulnerability testing
- Try many anomalous test cases

Mutation-based fuzzing
- Add anomalies to existing good inputs (e.g., test suite)

Generation-based fuzzing
- Generate inputs from specification of format, protocol, etc

Responsive fuzzing
- Leverage program instrumentation, code analysis
- Use response of program to build input set
Outline

› Introduction: static vs dynamic analysis
› Static analysis
  § Program execution using state descriptions
  § Security examples: static analysis to find vulnerabilities
› Dynamic vulnerability finding
  § Black-box testing: principles and web examples
  ❮ Fuzzing: principles and examples
Fuzzing examples

Discuss two examples in more detail

- PDF readers
- iPhone attack

Both by Charlie Miller, Independent Security Evaluators
PDF Readers

PDF format is extremely complicated

- PDF files can require complex rendering
  - Flash, Quicktime video, 3-d animation,
- PDF files can include executable JS
  - Extremely complicated code base
Mutation-based fuzzing with pdf

Similar process used for Acrobat, Preview (Apple)

- Collect a large number of pdf files
  - Aim to exercise all features of pdf readers
  - Found 80,000 PDF’s on Internet

- Reduce to smaller set with apparently equivalent code coverage
  - Used Adobe Reader + Valgrind in Linux to measure code coverage
  - Reduced to 1,515 files of ‘equivalent’ code coverage
  - Same effect as fuzzing all 80k in 2% of the time

- Mutate these files and test Acrobat (Preview) for vulnerabilities
Mutation

Modify each file in a number of ways

- Randomly change selected bytes to random values
- Produce ~3 million test cases from 1500 files
  - Approximately same numbers for Acrobat and Preview,
  - Even though code and methods for testing code coverage are different
- Use standard platform-specific tools to determine if crash represents a exploit
  - Acrobat: 100 unique crashes, 4 actual exploits
  - Preview: maybe 250 unique crashes, 60 exploits (tools may over-estimate)
Acrobat [Miller]

Mac tools
- Libgmalloc, dylib
- CrashWrangler

Windows tools
- !exploitable

Also tried
- Valgrind
- Many problems reported (not exploits)
› **Same Mac tools**
  - Libgmalloc, dylib
  - CrashWrangler

› **Valgrind**
  - Many problems reported (not exploits)
Comparison

Adobe Reader

100 crashes
30-40 unique
3-10 exploitable

Mac Preview

1373 crashes
230-280 unique
30-60 exploitable

Caveat: Several years ago; older versions of code
Was this enough fuzzing?

Shape of curve suggests more possible: keep fuzzing until diminishing returns
Second example: iPhone

Discuss briefly because method is different
- Source code for key component (browser) is available
- Developer unit testing data also available

Approach used to find attack against iPhone (several years ago):
- Survey known information about platform
- Select area where testing might have been incomplete
- Use fuzzing to cause crash
- Use crash information to see if attacker can control phone
Information used to plan strategy

› WebKit
  ▪ Most Apple Internet applications share the same code
  ▪ WebKit is an open source library (source code on svn)

› Test information available for JavaScript rendering

The JavaScriptCore Tests
If you are making changes to JavaScriptCore, there is an additional test suite you must run before landing changes. This is the Mozilla JavaScript test suite.

› Use test suite and code coverage (source code!) to see which portions of code might not be well tested
Initial steps: select potential “weak link”

- Build browser and instrument it using standard tool gcov
- Run test suit to determine code coverage
- Select target
  - Main JavaScript engine has 79.3% of its lines covered
  - Perl Compatible Regular Expression (PCRE) library is 17% of the overall code, 54.7% of its lines covered
  - Attack focused on PCRE
Find attack:

› Fuzz standalone PCRE code
  - Write regular expression fuzzer
    › Generated hundreds of regular expressions containing different number of “evil” strings: “[[**]]”
    › Some produced errors
      PCRE compilation failed at offset 6: internal error: code overflow.
    › Examined crash reports to find jump to registers that could be controlled

› Build attack using this information – Lots of Hard Work!
  - Select png file format and embed attack
  - Keep trying until attack works on actual phone
We discussed two fuzzing examples
- PDF readers: Acrobat and Preview
- iPhone vulnerability based on browser rendering png files

Some general themes
- Choosing good input is very important
  - Very unlikely to find significant crash by chance
  - Use many examples and simplify using information about system
- Standard tools can diagnose crash, find vulnerabilities
- Lots of hard work and only small change of success in many cases
  - But developer teams should use some of these methods too (easier for them!)
Announcement

› Guest lecture next Tuesday
› Paul Kocher
  ▪ Meltdown and Spectre – attacks that change everything