Autonomous Hacking: The New Frontiers of Attack and Defense
Hacking
Hacking Teams
Security compromises can be achieved through different routes
Hacking the User
Hacking the Process

CLICK HERE TO RESET YOUR PASSWORD TODAY!
Hacking the Code
Hacking Binary Code

- Low abstraction level
- No structured types
- No modules or clearly defined functions
- Compiler optimization and other artifacts can make the code more complex to analyze
- WYSIWYE: What you see is what you execute
Finding Vulnerabilities

Human | Semi-Automated | Fully Automated

RSAC Conference 2016
Manual Analysis

- “Look at the code and see what you can find”
- Requires substantial expertise
  - The analysis is as good as the person performing it
- Allows for the identification of complex vulnerabilities (e.g., logic-based)
- Expensive, does not scale
Tool-Assisted Analysis

- “Run these tools and verify/expand the results”
- Tools help in identifying areas of interest
  - By ruling out known code
  - By identifying potential vulnerabilities
- Since a human is involved, expertise and scale are still issues
Automated Analysis

- “Run this tool and find the vulnerability”
  - ... and possibly generate an exploit...
  - ... and possibly generate a patch
- Requires well-defined models for the vulnerabilities
- Can only detect the vulnerabilities that are modeled
- Can scale (not always!)
Automated Vulnerability Analysis

- An algorithm that takes as input a code artifact (source code, byte-code, binary code) and identifies potential vulnerabilities.

- The Halting Problem: “the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running or continue to run forever.”


- Alan Turing proved that a general algorithm does not exist.
Types of Vulnerability Analysis

- **Static Analysis**
  - A form of abstract interpretation
  - Does not execute the code

- **Dynamic Analysis**
  - A form of concrete interpretation
  - Executes the code (or a model of it)
The goal of static analysis techniques is to characterize all possible run-time behaviors over all possible inputs without actually running the program.

Find possible bugs, or prove absence of certain kinds of vulnerabilities.

Static analysis has been around for a long while.
- Type checkers, compilers
- Formal verification

Challenges: soundness, precision, and scalability.
Soundness and Completeness

Actual run-time behaviors

$P$
Soundness and Completeness

Over-approximation (sound)

Actual run-time behaviors

$P$
Soundness and Completeness

More precise over-approximation (sound)

Actual run-time behaviors

\( P \)
Soundness and Completeness

- Actual run-time behaviors
- Under-approximation (complete)
Soundness and Completeness

Actual run-time behaviors

Unsound, incomplete analysis

$P$
Example Analyses

- Control-flow analysis: Find and reason about all possible control-flow transfers (sources and destinations)
- Data-flow analysis: Reason about how data flows at run-time (from sources to sinks)
- Data dependency analysis: Reason about how data influences other data
- Points-to analysis: Reason about what values can pointers take
- Alias analysis: Determine if two pointers might point to the same address
- Value-set analysis: Reason about what are all the possible values that variables can hold
Dynamic Analysis

- Dynamic approaches are very precise for particular environment and inputs
  - You execute the code!
- However they provide no guarantee of coverage
  - You evaluate only the part of a program that you exercise!
Example Analyses

- Taint analysis
- Fuzzing
- Forward symbolic execution
- Concolic execution
Fuzzing

- Fuzzing is an automated procedure to send inputs and record safety condition violations as crashes
  - Assumption: crashes are potentially exploitable

- Several dimensions in the fuzzing space
  - How to supply inputs to the program under test?
  - How to generate inputs?
  - How to generate more “relevant” crashes?
  - How to change inputs between runs?

- Goal: maximized effectiveness of the process
Gray/White-box Fuzzing

- **Input Generator**
- **Application Under Analysis**
- **Fuzzing Infrastructure**
- **Crash**
- **Crash Database**
- **Feedback**
- **Bugs (0-day)**
Fuzzing: American Fuzzy Lop
Fuzzing: American Fuzzy Lop

- Instrumentation-guided genetic fuzzer developed by Michael Zalewski

- The instrumentation collects information at branch points
  - Supports the generation of inputs that improve coverage

- Inputs that bring new paths are considered more interesting and queued for further exploration

- Inputs are chosen and mutated

- Unique crashes are identified using branch analysis (instead of stack summaries)
Symbolic Execution: angr

- Framework for the analysis of binaries
- Supports a number of architectures
  - x86 (32 and 64), MIPS, ARM, PPC, etc.
- http://angr.io
- https://github.com/angr
- angr@lists.cs.ucsb.edu
angr Components

- Binary Loader
- Static Analysis Routines
- Symbolic Execution Engine
  - Control-Flow Graph
  - Data-Flow Analysis
  - Value-Set Analysis
  - Forward Symbolic Execution
  - Under-constrained SE
Symbolic Execution

- "How do I trigger path X or condition Y?"

Dynamic analysis
- Input A? No. Input B? No. Input C? ...
- Based on concrete inputs to application

(Concrete) static analysis
- You can’t/You might be able to
- Based on various static techniques
Symbolic Execution

- "How do I trigger path X or condition Y?"
- Interpret the application
- Track "constraints" on variables
- When the required condition is triggered, "concretize" to obtain a possible input
Concretization

- Constraint solving
- Conversion from set of constraints to set of concrete values that satisfy them

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt;= 10</td>
</tr>
<tr>
<td>x &lt; 100</td>
</tr>
</tbody>
</table>

Concretize: x = 42
Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```
Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

State A

<table>
<thead>
<tr>
<th>Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = ???</td>
<td>------</td>
</tr>
</tbody>
</table>
Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

**State A**

- **Variables**
  - `x = ???`
- **Constraints**
  - `x >= 10`

**State AA**

- **Variables**
  - `x = ???`
- **Constraints**
  - `x < 10`

**State AB**

- **Variables**
  - `x = ???`
- **Constraints**
  - `x >= 10`
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

**State AA**
- **Variables**
  - `x = ???`
- **Constraints**
  - `x < 10`

**State AB**
- **Variables**
  - `x = ???`
- **Constraints**
  - `x >= 10`

**State ABA**
- **Variables**
  - `x = ???`
- **Constraints**
  - `x >= 10`
  - `x < 100`

**State ABB**
- **Variables**
  - `x = ???`
- **Constraints**
  - `x >= 10`
  - `x >= 100`
Example

```python
x = int(input())
if x >= 10:
    if x < 100:
        vulnerable_code()
    else:
        func_a()
else:
    func_b()
```

State ABA

<table>
<thead>
<tr>
<th>Variables</th>
<th>x = ???</th>
</tr>
</thead>
</table>

Constraints

| x >= 10 |
| x < 100 |

Concretized ABA

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 99</td>
</tr>
</tbody>
</table>
Putting It All Together

- Fuzzing excels at producing general input
- Symbolic execution is able to satisfy complex path predicates for specific input
- Key insight: Combine both techniques to leverage their strengths and mitigate their weaknesses
Assisting Fuzzing with Symbolic Execution

Fuzzing

- good at finding solutions for general input

Symbolic Execution

- good at finding solutions for specific input
Driller

Test Cases
“Cheap” fuzzing coverage
Driller

“Cheap” fuzzing coverage

Dynamic Symbolic Execution

Test Cases

“Χ”

“Η”
Driller

“Cheap” fuzzing coverage

Dynamic Symbolic Execution

New test cases generated

Test Cases

“Χ”

“𝑌”

“CGC_MAGIC”
Driller

“Cheap” fuzzing coverage

Dynamic Symbolic Execution

New test cases generated

Test Cases

“X”

“Y”

“CGC_MAGIC”

“CGC_MAGICY”
Why Hacking?

- Vulnerability analysis can be used
  - Offensively
  - Defensively
  - For fun (and profit)

- Hacking competitions have become a popular venue for the application of breakthrough techniques in vulnerability analysis
  - DefCon CTF
  - Pwn20wn
Many Competition Styles

**Challenge-based**
- Should not be called “CTF”!
- Easy to organize
- Easy to scale
- Exclusively focused on attacking
- No real-time component

**Interactive, online CTFs**
- Very difficult to organize
- Require substantial infrastructure
- Difficult to scale
- Focused on both attacking and defending in real time
Current Interactive, Online CTFs

- From ctftime.org: 100+ events listed
- Online attack-defense competitions:
  - UCSB iCTF 13 editions
  - RuCTF 5 editions
  - FAUST 1 edition
The iCTF Framework

- Lessons learned from running iCTFs were the basis for building a framework
- The framework formalizes the structure of services and allows for the reuse of the infrastructure
- Available at:
  - http://github.com/ucsb-seclab/ictf-framework
  - http://ictf.cs.ucsb.edu/framework
CTFs Are Playgrounds...

- For people (hackers)
- For tools (attack, defense)
- But can they be used to advance science?
DARPA Competitions

Self-driving Cars

Robots
The DARPA Cyber Grand Challenge Programs!
The DARPA Cyber Grand Challenge
The DARPA Cyber Grand Challenge
The DARPA Cyber Grand Challenge

- CTF-style competition
- Autonomous Cyber-Reasoning Systems (CRS) attack and defend a number of services (binary programs)
- NO HUMAN IN THE LOOP
- A first qualification round decided who the 7 finalists are
  - Qualification comes with a $750,000 cash prize
- The final event is scheduled for August 4, 2016 during DefCon
  - The top team will receive a $2,000,000 cash prize
Shellphish CGC Team
CGC Other Finalists

CodeJitsu

CSDS

DeepRed

disekt

ForAllSecure

TECHx
The CGC Environment

- Binaries run on a custom OS, called DECREE
  - Limited number of system calls
- A POV has to demonstrate the ability:
  - To read a specific value from memory
  - To set a register to a specific value
- Not all rules have been finalized
The Shellphish CRS: ShellWePlayAGame?
The Shellphish CRS: ShellWePlayAGame?

- PCAP
- CB
- Autonomous processing
- Autonomous vulnerability scanning
- Autonomous patching
- Test cases
- Proposed POVs
- Proposed RBs
- Autonomous service resiliency
- POV
- RB
- Proposed POVs

63
May the Best CRS Win!

\[ \sum_{i=0}^{CB} \text{Availability} \times \text{Security} \times \text{Evaluation} \]

- Patching cannot affect performance
- Patching cannot affect functionality
- When you are shooting blindfolded automatic weapons, it’s easy to shoot yourself in the foot...
The goal of the CGC is to foster the development of new attack and defense techniques that...
- Automatically identify and exploit vulnerabilities in binary programs
- Automatically patch vulnerability and provide functionally-equivalent yet secure versions of a vulnerable binary
What Does All This Mean to YOU?

- Novel automated analysis techniques will allow for
  - The identification of vulnerabilities (and, possibly, backdoors) in binaries before they are deployed
  - The patching of binaries on-the-fly without having to wait for vendors’ fixes
  - Scale...
What Can I Do NOW?

- Use CTFs (or other security competitions) to foster computer education in your company
- The iCTF Framework is free, open, and can be used to create sophisticated attack-defense security competition within your organization
- Familiarize yourself with vulnerability analysis tool and learn how to use them as integral part of your development process
- After all...
Human + Machine = WIN!

OMG, can’t do stairs?!?
Extra Slides
The iCTF Architecture

- **VPN Server**
  - Team 1
  - Team 2
  - Team 21
  - Team 22

- **Vulnerable Server**
  - 10.7.1.2
  - 10.7.2.2
  - 10.7.21.2
  - 10.7.22.2

- **VPN Gateway**
  - 10.7.1.X

- **Team Interface**
- **Scoreboard**
  - Shows the current scores to the teams
  - Allows teams to register and submit flags

- **Database**
  - Stores the state of the competition

- **Gamebot**
  - For each tick, schedules scripts and compute scores

- **Scriptbot**
  - Runs the scripts to update flags and check for services
Example: Simple Overflows

```c
int main(int argc, char** argv) {
    char buf[256];
    strcpy(buf, argv[1]);
    return 0;
}
```

- Simplest detection approach: grep for `strcpy`
- More rigorous:
  - Determine data flow from command-line argument to `strcpy`'s parameter
  - Determine size of source, destination buffers
  - Model semantics of `strcpy`
  - Check safety condition: `len(argv[1]) < len(buf)`
Fuzzing vs. Symbolic Execution

Percent of Transitions Found as Iterations of Concolic Execution and Fuzzing

- Black bars represent concolic execution.
- Gray bars represent fuzzing.

Example labels: 24d9501_01, 549bc01_01, etc.
Exploration Process
# The UCSB iCTFs

<table>
<thead>
<tr>
<th>Year</th>
<th>Theme</th>
<th>Teams</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>Open-Source Windows</td>
<td>7</td>
</tr>
<tr>
<td>2004</td>
<td>UN Voting System</td>
<td>15</td>
</tr>
<tr>
<td>2004</td>
<td>Bass Tard Corporation</td>
<td>9</td>
</tr>
<tr>
<td>2005</td>
<td>Spam Museum</td>
<td>22</td>
</tr>
<tr>
<td>2006</td>
<td>Hillbilly Bank</td>
<td>25</td>
</tr>
<tr>
<td>2007</td>
<td>Copyright Mafia</td>
<td>36</td>
</tr>
<tr>
<td>2008</td>
<td>Softterror.com Terrorist Network</td>
<td>39</td>
</tr>
<tr>
<td>2009</td>
<td>Rise of the Botnet</td>
<td>57</td>
</tr>
<tr>
<td>2010</td>
<td>Rogue Nation of Litya</td>
<td>73</td>
</tr>
<tr>
<td>2011</td>
<td>Money Laundering</td>
<td>89</td>
</tr>
<tr>
<td>2012</td>
<td>SCADA Defense</td>
<td>92</td>
</tr>
<tr>
<td>2013</td>
<td>Nuclear Cyberwar</td>
<td>123</td>
</tr>
<tr>
<td>2014</td>
<td>Large-Scale Hacking</td>
<td>86</td>
</tr>
<tr>
<td>2015</td>
<td>Crowdsourced Evil</td>
<td>35</td>
</tr>
</tbody>
</table>
Branch Tracking

- The instrumentation collects information about which branches are taken.

- The information is stored in a shared hash table. A branch from a previous location to the current location triggers the instrumentation code:

  ```
  cur_location = <COMPILE_TIMERANDOM>;
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
  ```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;

- Note that the index in the hash is a combination of the previous and current location

- The size of the shared memory is 64K
  - Big enough to avoid collisions
  - Small enough to be fast and fit in memory caches

- The shift of the marker for the current location allows for
  - Distinguishing A->B from B->A
  - Distinguishing A->A from B->B
Branch Tracking

- Branch tracking is a better metric for program exploration than plain basic block coverage.

- Consider the following cases, where A, B, C, D, E are code blocks:
  - A -> B -> C -> D -> E (tuples: AB, BC, CD, DE)
  - A -> B -> D -> C -> E (tuples: AB, BD, DC, CE)

- While the same amount of code is covered, but different paths are taken.
Guiding Exploration

- AFL maintains a global map of all the paths observed in all the executions up to the current one.
- When a mutated input file introduces tuple that were not observed before, the input file is queued for further processing.
- Inputs that do not generate new transitions, are discarded (even if the sequence has not been seen before).
Example

- #1: A -> B -> C -> D -> E
- #2: A -> B -> C -> A -> E (C->A, A->E are new)
- #3: A -> B -> C -> A -> B -> C -> A -> B -> C -> D -> E (no new tuples)
Counting Branches

- AFL keeps track of how many times a certain transition happens for each run
  - Buckets: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+

- If a particular input causes a transition to move between buckets, then the input is deemed interesting and queues for processing
  - Buckets allow for emphasizing small changes (1 to 2) vs. not-so-relevant changes (67 to 70)
Processing Input

- The interesting file (thousands) are added to the input queue
  - Usually 10-30% from the discovery of new transitions
  - The rest from changes in the hit count

- The input queue is analyzed so that a subset of the (best) files is marked as “favorite”
  - The files cover all the tuples
  - The files have lowest latency and size
Prioritizing the Inputs

1. Choose a tuple from the ones observed so far and put it in a set
2. Select the input that caused the shortest execution and has the smallest size
3. Add all the transitions observed for that execution to the set
4. If the set does not covered all the previously observed transitions, goto 1
Prioritizing the Inputs

- If there are new, yet-to-be-fuzzed favorites present in the queue, 99% of non-favored entries will be skipped to get to the favored ones.

- If there are no new favorites:
  - If the current non-favored entry was fuzzed before, it will be skipped 95% of the time.
  - If the current non-favored entry was not fuzzed before, the odds of skipping it are 75%.

- These values are chosen to balance queue cycling and diversity.
Fuzzing Strategies

- Sequential bit flips with varying lengths and stepovers
- Sequential addition and subtraction of small integers
- Sequential insertion of known interesting integers (0, 1, INT_MAX, etc.)
- Stacked bit flips, insertions, deletions, arithmetic operations, and splicing of different test cases
- It is also possible to provide dictionaries of known keywords to help in the fuzzing process