Sidewinder

An Evolutionary Guidance System For Malicious Input Crafting
Software Vulnerability

• Refers to a weakness in a system allowing an attacker to violate the integrity, confidentiality, access control, availability, consistency or audit mechanism of the system or the data and applications it hosts (Wikipedia)

• May exist only in theory or have a working exploit
Potential vs. Exploitable

- Potential vulnerabilities - locations within a program that contain known weaknesses
  - Ex. The usage of APIs known to be susceptible to buffer overflows
  - Potential vulnerabilities may or may not be exploitable

- Exploitable vulnerabilities - exist when a potentially vulnerable program location...
  - Is dependent on or able to be influenced by user supplied input
  - Is reachable on the program control flow graph at runtime
White Box Analysis

• Also known as “glass box testing” or “structural testing”

• Involves detailed, manual, static analysis of source or disassembly to gain understanding of internal program structure

• Pros
  - Human mind is good at pattern recognition and is better at uncovering subtle bugs unlikely to be located with automated tools

• Cons
  - Time consuming (and thus costly)
  - Sometimes difficult to tell whether a potential vulnerability that is dependent upon external input will be reachable at runtime
Black Box Analysis

- Also known as “concrete box testing” or “functional testing”

- Does not rely on human understanding of source or disassembly
  - Involves injecting random or semi-random input into a program and monitoring output for unexpected behavior

- Pros
  - Easily automated
  - Vulnerabilities discovered at runtime are definitely reachable and the input structure that caused them is known

- Cons
  - Random nature of input space exploration makes the probability of discovering vulnerabilities highly non deterministic
Black Box Analysis
(Fuzzers)

• Fuzzers - inject malformed input into a program and then monitor it for crashes

• Many bugs are the result of programmer oversights or assumptions regarding the structure of user supplied input
  - Often used to find bugs in parser / protocol handling logic

• Examples:
  - **Spike**: A collection of many fuzzers from Immunity
  - **File Fuzz**: A file format fuzzer for PE (Windows) binaries from iDefense.
  - **Peach Fuzz**: Framework for building fuzzers written by Michael Eddington
Fuzzers: The Good & Bad

• The Good
  - Fully automated software attacks
  - Random or pseudo random input selection results in widely sampling the input space
  - May generate test inputs that a human wouldn’t think of

• The Bad
  - Most fuzzers aren’t very intelligent
    • We don’t learn anything from past inputs that can help us select better test inputs in the future!
  - No good measurement of attack progress
    • The program either crashes or it doesn’t!
  - Nondeterministic time frame for finding an interesting bug
    • The program has an equal likelihood of crashing 2 minutes from now or 2 weeks from now!
Smarter Fuzzers

**Goals**

1. To have the fuzzer learn something from past inputs that it can use to improve input selection in the future.

2. To improve the odds of finding something interesting within a reasonable time frame.
   - That is, we should use the knowledge we gain from past experience to preferentially drive the program toward states that have a greater potential for vulnerability.

3. To keep the attack automated as much as possible.
What to learn? (1)

- The runtime execution trace is dependent upon both user-supplied input and the static structural characteristics of the program control flow graph.

- Normal fuzzers have no measurement of *how much* or *what portions* of the program and input state spaces have been explored in the past.

- If we had this information, maybe we could use it to choose better inputs?
Consider...

- Greater code coverage may correlate to greater chance of discovering a vulnerable program state

- By linking inputs with their runtime execution paths, we may be able to select for inputs that will have a greater likelihood of taking *specific, dependent execution paths that lead to potentially vulnerable states*

- **Example:** Paths to basic blocks indicating usage of API’s known to be susceptible to buffer overflows or format string vulnerabilities
An Input Crafting Problem

- What does the input have to look like for us to exercise the code path between input node (recv) & the potentially vulnerable node (strcpy)???

Parsing & validation logic on path between recv and strcpy
How?

- We can disassemble the program and manually decode the packet parsing logic (white box).
- We can throw random inputs at it hoping one will eventually get the strcpy we think might be vulnerable (black box).
- Or we can try to do a little better...
A Search Problem?

- What if we could automatically decode the packet parsing logic? Or at least *evolve* an approximation heuristically?

- Can we model input crafting as a generalized search problem?
  - That is, aren’t we in some sense searching for those inputs that conform to a structure capable of taking specific, dependent execution paths that lead to portions of a program with a higher than average likelihood of vulnerability?

- We can perform this search by driving input selection using a *genetic algorithm* where the relative “fitness” or “goodness” of a specific input is related to its progress on the program control flow graph.
The Basic Idea...

- Over time, some inputs will be better than others:
  - They increase code coverage by reaching previously unexplored areas of the CFG
  - They are on a path to a basic block where some potentially vulnerable API is being used

- If we “mate” the best of the inputs we’ve found in the past...
  - We can select for those characteristics in the future that maximize code coverage and drive inputs down execution paths with potential vulnerabilities.
First a little theory...
Genetic Algorithms

- A type of algorithm that mimics evolution

- What is an algorithm?
  - Specific set of steps to find a solution to a specific type of problem

- What is evolution?
  - Natural process which acts on a population of organisms
  - Hereditary information is passed from one generation to the next in the organism’s genome
  - Mutation adds random variation to the genome
  - Natural selection removes organisms whose genetic code is less fit for their environment
  - With each passing generation, the organisms in the population are better suited to their environment
Genetic Algorithms

- Genetic algorithms are stochastic global optimizers
  - Random component of the algorithm, so it won’t run the same way twice
  - Finds better solutions, but may not find the best, *even if you run it forever*

- **Example**: Maximizing the number of ones in a binary string of length 10
Genetic Algorithms

- Requires three things
  - A representation
    - What solutions to the problem look like (its genome)
  - A fitness function
    - An equation that operates on a solution and tells you how good or bad it is
  - Genetic operators
    - Mutation and crossover

Example:
- Representation: 10 digit binary string
- Fitness function: the number of ones
Genetic Algorithms

- It works like this:
  1. Start out with a *population* of random solutions
  2. Calculate each solution’s *fitness*
  3. Select solutions with highest fitness
  4. Slightly *mutate* the selected solutions and then perform *crossover* (mating)
  5. Create the next *generation* from offspring and then go to step 2.
Step 1: Initial Population

- Start out with a *population* of random solution *genomes* in the chosen *representation*.

- **Example**: Create 4 random binary strings

  Population

  0100100000
  1000001010
  1110100111
  0000001000
Step 2: Calculate Fitness

- Calculate the *fitness function* for each member of the *population’s genome*

- **Example:** Count the number of ones in each string

<table>
<thead>
<tr>
<th>Population</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100100000</td>
<td>2</td>
</tr>
<tr>
<td>1000001010</td>
<td>3</td>
</tr>
<tr>
<td>1110100111</td>
<td>7</td>
</tr>
<tr>
<td>0000001000</td>
<td>1</td>
</tr>
</tbody>
</table>
Step 3: Selection

- Find out which solutions are fittest and ignore the rest

**Example:** The genomes having fitness 3 and 7 are the fittest

<table>
<thead>
<tr>
<th>Population</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100100000</td>
<td>2</td>
</tr>
<tr>
<td>1000001010</td>
<td>3</td>
</tr>
<tr>
<td>1110100111</td>
<td>7</td>
</tr>
<tr>
<td>0000001000</td>
<td>1</td>
</tr>
</tbody>
</table>
Step 4a: Crossover

- Create new genomes by randomly swapping their genomes at a random point

- **Example:** Use the two genomes we selected in the previous slide and swap at location 3

<table>
<thead>
<tr>
<th>Parent 1</th>
<th>Parent 2</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000001010</td>
<td>1110100111</td>
<td>1110100111</td>
</tr>
<tr>
<td>1110100111</td>
<td>1000101010</td>
<td>1000100111</td>
</tr>
</tbody>
</table>
Step 4a: Crossover

- Create new genomes by randomly swapping their genomes at a random point
- **Example**: Use the two genomes we selected in the previous slide and swap at location 6
Step 4b: Mutation

- Inject more variation into the *population* by randomly flipping a bit with a certain low probability

- **Example:** Flip bits at random in the offspring we generated

<table>
<thead>
<tr>
<th>Population</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110100111</td>
<td>1100100111</td>
</tr>
<tr>
<td>1000100111</td>
<td>1000100111</td>
</tr>
<tr>
<td>1110101010</td>
<td>1110111010</td>
</tr>
<tr>
<td>1000000111</td>
<td>1001000111</td>
</tr>
</tbody>
</table>
Step 5: GOTO 2

- We now have a the next generation, a new population we treat just like the previous one.

- **Example:** We count the ones again. On average, they have slightly higher fitness.

<table>
<thead>
<tr>
<th>Population</th>
<th>Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100100111</td>
<td>6</td>
</tr>
<tr>
<td>1000100111</td>
<td>5</td>
</tr>
<tr>
<td>1110101110</td>
<td>7</td>
</tr>
<tr>
<td>1001000111</td>
<td>5</td>
</tr>
</tbody>
</table>
A Note On Mutation & Crossover Rates

- The goal is to strike a balance between preserving existing information and generating new information...
  - Crossover preserves information
  - Mutation generates information
    - High mutation rate → aggressive, global exploration of search space
    - Low mutation rate → less aggressive, local exploration of search space

- Static or dynamic?
  - Dynamic mutation rates adjust according to the current progress of the search. Static ones do not.
  - e.g. We may choose to raise the mutation rate if our candidate solutions are not improving in fitness after some set amount of time
Two Things We Need...

- A **representation**
  - What input are we going to inject?

- A **fitness function**
  - How are we going to measure how good the input is?
We need to inject input in a certain format (e.g. valid packet format in a parsing program)

Our representation describes the steps used to build the input string
- The benefit of evolving steps (as opposed to evolving the strings themselves) is that we can preserve some description of the dependency between user input and program structure
- Enables us to potentially “learn” how to approximate a valid input format without apriori knowledge (applicable to parser code)

We use a special kind of rule set called a context-free grammar
Context Free Grammars

• Consists of:
  - *Terminals* - the characters in the language
  - *Nonterminals* - place holders, much like variables in algebra
  - *Production rules* - substitutions you can make for each nonterminal
  - *Initial rule* - the first production rule, where the whole thing beings
Example

Initial Rule

S → sAs | xBx | m
A → bBb | B
B → aAa | C | AB
C → c | d | e

Production rules

Nonterminals

Terminals
Example

S → sAs | xBx | m
A → bBb | B
B → aAa | C | AB
C → c | d | e

S → xBx → xaAax → xabBbax → xabCbax

xabdbax
A grammar is a description of how to build all the strings

Our representation is a string of integers

How do we use the grammar to build a string in the language?

How do we turn 10247 into xabdbax?
Grammatical Evolution

- To produce a string from in our grammar using a series of integers, we use *grammatical evolution*, which can be summarized in pseudocode:

```plaintext
while (nonterminals in the string) {
    find first nonterminal;
    numRules = number of production rules for first nonterminal
    i = (next integer in the genome) % numRules;
    apply productionRule[i];
}
```
Grammatical Evolution

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>sAs</td>
<td>xBx</td>
</tr>
<tr>
<td>A</td>
<td>bBb</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>aAa</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

S → xBx → xaAax → xabBbax → xabCbax

xabdbax
Two Things We Need...

✔ A representation
  - Grammatical evolution

☒ A fitness function
  - How are we going to measure how good the input is?
• We can observe the program’s dynamic behavior and orient ourselves with the static control flow graph

• We want inputs that maximize code coverage
  - In other words, inputs that cause previously unobserved behavior
  - In other other words, inputs that go places on the control flow graph previous inputs haven’t explored
• Statistical models are handy for explaining what we mean by “rare” in a quantifiable way

• A particular type of statistical model, called a *Markov process*, is appropriate here

• Rather than bore you with theory, I’ll try to show you how they work
During each *generation* of the genetic algorithm, we keep a running total (or *sample*) of the solutions that used each transition in the control flow graph.
To compute the fitness of a solution, we simply calculate its probability assuming a Markov process from the sampled results (lower is better).

Path = A, C, E, D, G, M

Fitness = \[ 0.75 \times 0.9 \times 0.5 \times 0.67 \times 0.8 = 0.18 \]
Two Things We Need...

- A **representation**
  - Grammatical evolution

- A **fitness function**
  - Sampled Markov Process
Implementation: Extracting The Program CFG

- We extract subgraph of overall CFG that includes all nodes existing on a path between input acceptance node and target nodes (potentially vulnerable nodes containing things like strcpy calls)
  - Use IDA’s plugin SDK to construct graph
  - Nodes with edges directed outside subgraph are placed within a “rejection set”.
• Identify source (input) and a destination (potentially vulnerable) nodes
• Identify all nodes on a path between source and target nodes
• Identify reject nodes
  - i.e. the nodes that bound a known path to the target but do not exist on a path themselves
Instrumenting the program CFG

- We place breakpoints on the entry points for all extracted subgraph nodes.
  - They are used to evaluate progress on the runtime execution path for a given input
  - The execution path is tracked until a rejection node is reached (i.e. the destination is no longer reachable along all subsequent execution paths) OR target node has been reached
  - When the destination is determined to be no longer reachable, but we have not yet reached the target nodes we stop and try the next input
Illustration

Breakpoints

source

recv

destination

strcpy

Path Nodes

Reject Nodes

Breakpoints
Evolving program input

- Starting with an initial population
  - Run each input through the program and track execution path. If program crashes, log it and restart.
  - Calculate “fitness” of each input based upon its path
  - Choose the “fittest” individuals and mate them to form the next population of inputs
  - Run new inputs until target node has been successfully reached.
Advantages

- We apply knowledge gained from past experience to drive our choice for future inputs
  - Well suited to applying to parser code, which has a rich control flow structure for the GA to learn from

- Minimal knowledge of input structure required
  - GA can learn to approximate input format during execution

- Once a target location has been reached, the algorithm continues to exploit weaknesses in the CFG to produce additional, different inputs capable of reaching it
Limitations

- Difficulty to extract some parts of the CFG statically
  - Thread Creation
  - Call tables

- Dependent upon CFG structure
  - Program must have enough information embedded within its structure for the GA to be able to “learn from”
    - Assumes dependency between graph structure and user supplied input (an example would be parser code)
  - Not useful for programs that have a ‘flat’ CFG structure
  - Finding all paths has high complexity $O()$ and takes a long time on large program graphs
  - We can prove reachability by getting to a potentially vulnerable target state, but failure to get there does not mean the location is unreachable!
Conclusions

• Shows how genetic algorithms can be applied to the external input crafting process to maximize exploration of program state space and intelligently drive a program into potential vulnerable states.

• Automated approach → treats the internal structure of each node in the CFG as a black box.

• Needs testing on more complex programs
  - Our work is theoretical and prototypish

• Needs testing on more complex programs
To Summarize ;)