

# Sidewinder

*An Evolutionary Guidance System For Malicious Input Crafting*



August 2006

Shawn Embleton  
Sherri Sparks  
Ryan Cunningham



# 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 weather 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...

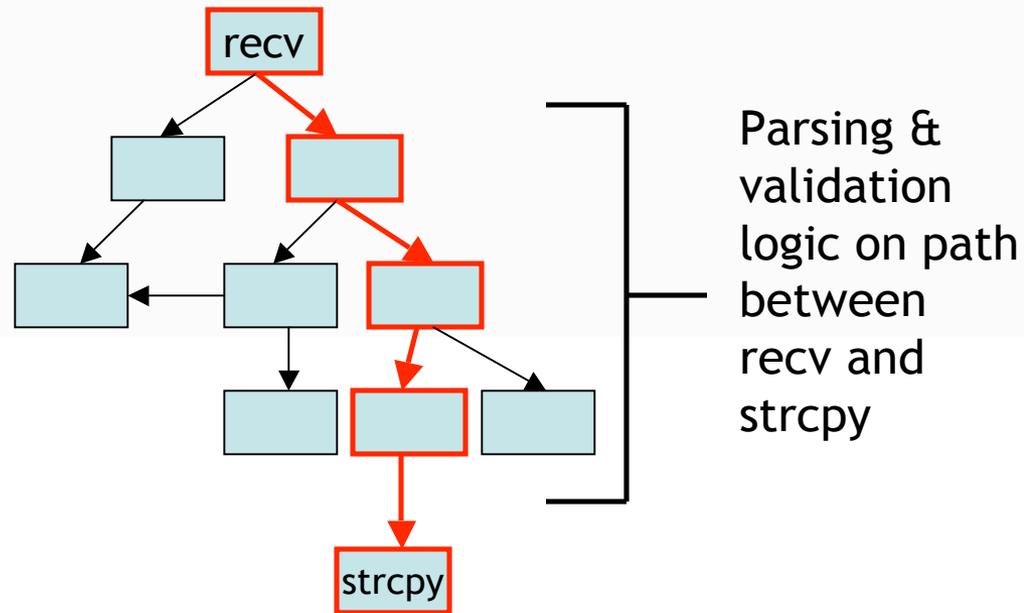


- Greater code coverage may correlate to greater chance of discovering a vulnerable program state
- By linking inputs with their runtime execution paths, we may be able to select for inputs that will have a greater likelihood of taking *specific, dependent execution paths that lead to potentially vulnerable 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) ???





# 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



Population	Fitness
0100100000	2
1000001010	3
1110100111	7
0000001000	1



# Step 3: Selection

- Find out which solutions are fittest and ignore the rest
- **Example:** The genomes having fitness 3 and 7 are the fittest



Population      Fitness

0100100000      2

1000001010      3

1110100111      7

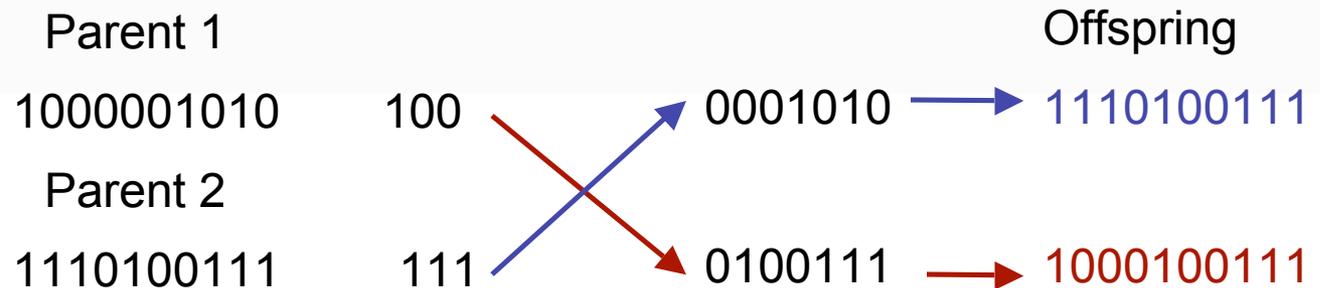
0000001000      1



# 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

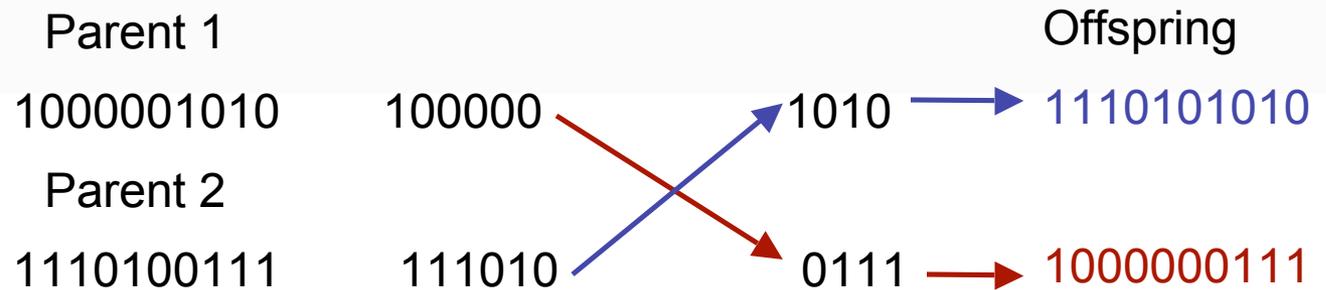




# Step 4a: Crossover



- 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

Population

1110100111	11 <b>0</b> 0100111
1000100111	1000100111
1110101010	11101 <b>1</b> 1010
1000000111	100 <b>1</b> 000111



# 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.

Population	Fitness
1100100111	6
1000100111	5
1110101110	7
1001000111	5



# 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?





# Representation



- 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



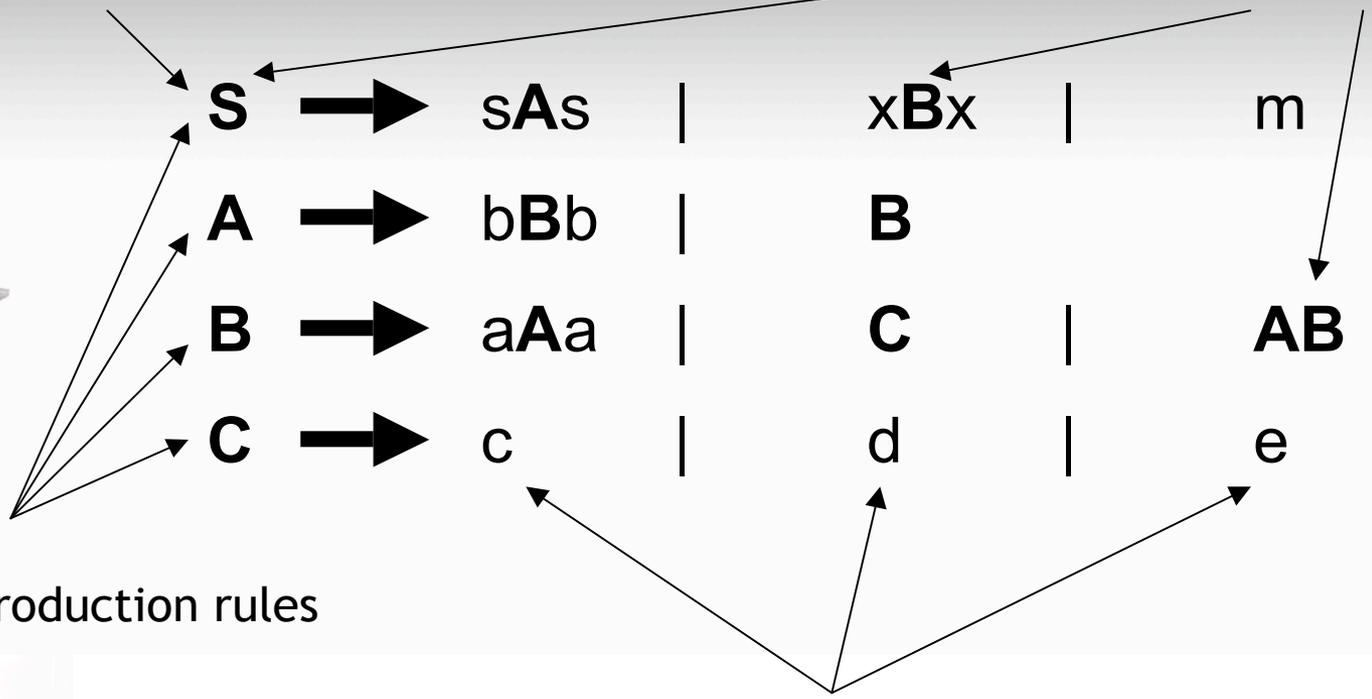
Initial Rule

Nonterminals

<b>S</b>	→	sAs		xBx		m
<b>A</b>	→	bBb		<b>B</b>		
<b>B</b>	→	aAa		<b>C</b>		<b>AB</b>
<b>C</b>	→	c		d		e

Production rules

Terminals





# Example



<b>S</b>	→	sAs		xBx		m
<b>A</b>	→	bBb		<b>B</b>		
<b>B</b>	→	aAa		<b>C</b>		<b>AB</b>
<b>C</b>	→	c		d		e

**S** → **xBx** → **xaAax** → **xabBbax** → **xabCbox**

xabdbax

A thick black arrow pointing from the 'x' in 'xabCbox' down and to the left towards 'xabdbax'.



# More on Representation



- 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:

```
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];  
}
```

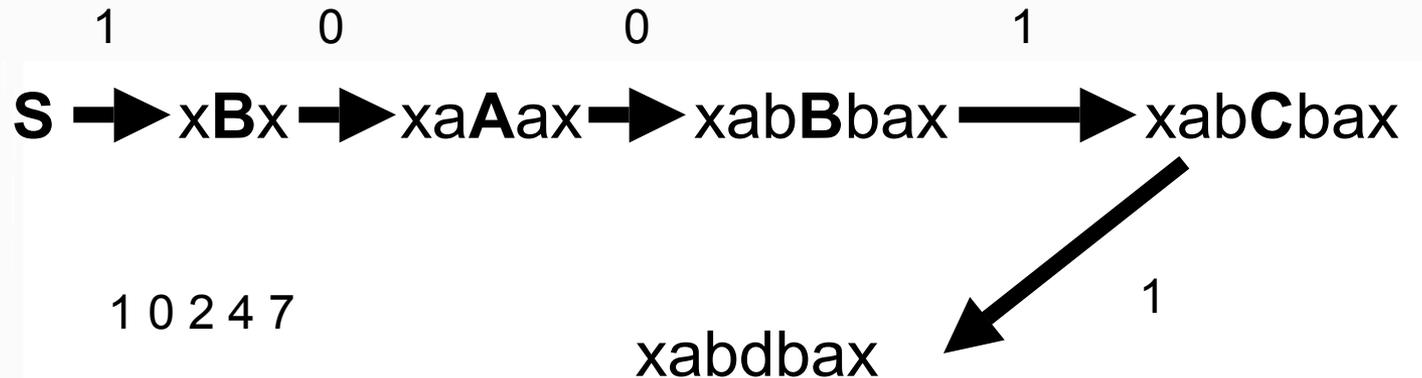




# Grammatical Evolution



	<i>0</i>	<i>1</i>	<i>2</i>
<b>S</b> →	s <b>A</b> s	x <b>B</b> x	m
<b>A</b> →	b <b>B</b> b	<b>B</b>	
<b>B</b> →	a <b>A</b> a	<b>C</b>	<b>AB</b>
<b>C</b> →	c	d	e





# Two Things We Need...

✓ *A representation*

- Grammatical evolution

□ *A fitness function*

- How are we going to measure how good the input is?





# Fitness Function



- 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



# Markov Process



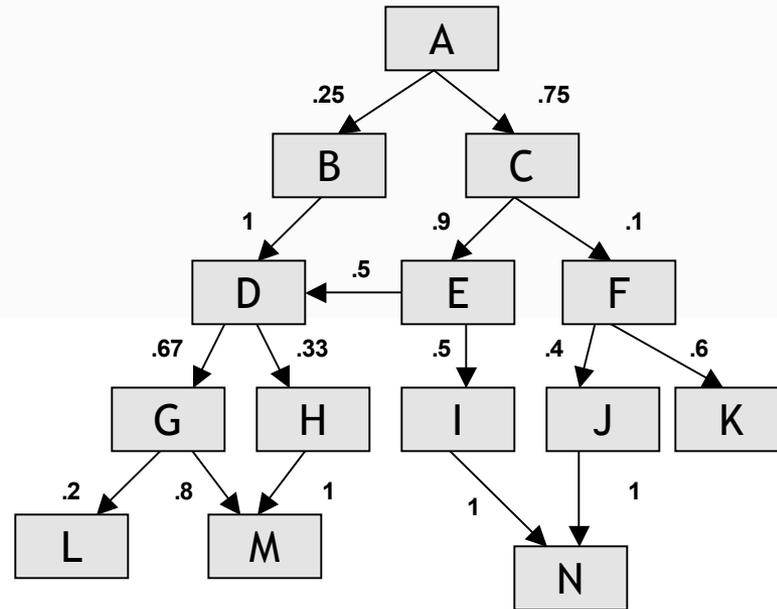
- Statistical models are handy for explaining what we mean by “rare” in a quantifiable way
- A particular type of statistical model, called a *Markov process*, is appropriate here
- Rather than bore you with theory, I’ll try to show you how they work



# Markov Process Example



- 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

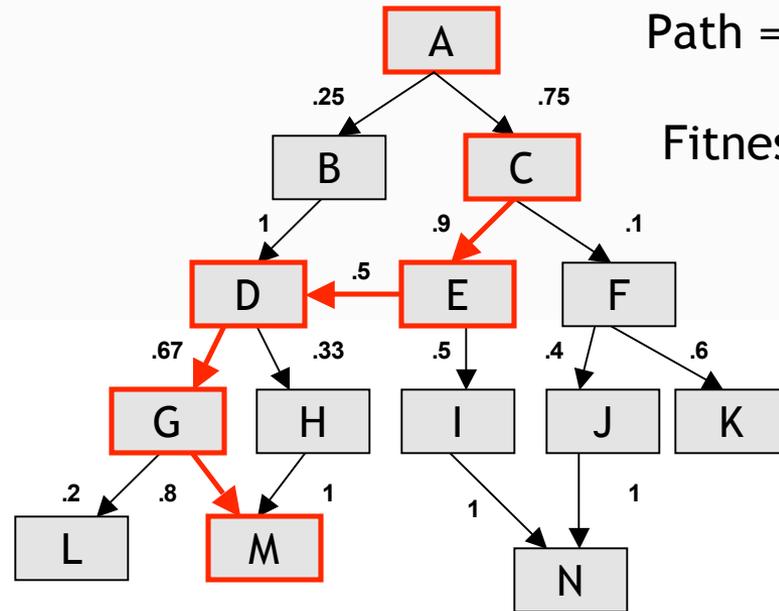




# Markov Process Example



- 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

$$\text{Fitness} = .75 \times .9 \times .5 \times .67 \times .8 = .18$$



# Two Things We Need...

- ✓ ***A representation***
  - Grammatical evolution
- ✓ ***A fitness function***
  - Sampled Markov Process





# Implementation: Extracting The Program CFG



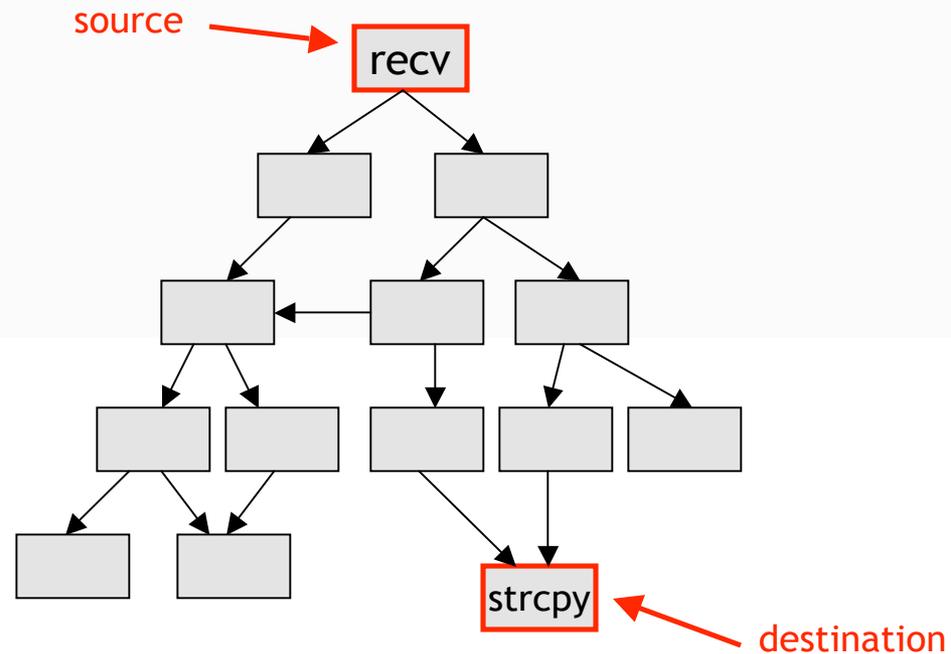
- We extract subgraph of overall CFG that includes all nodes existing on a path between input acceptance node and target nodes (potentially vulnerable nodes containing things like strcpy calls)
  - Use IDA's plugin SDK to construct graph
  - Nodes with edges directed outside subgraph are placed within a "rejection set".



# Illustration

## Extracting The Program CFG (1)

- Identify source (input) and a destination (potentially vulnerable) nodes

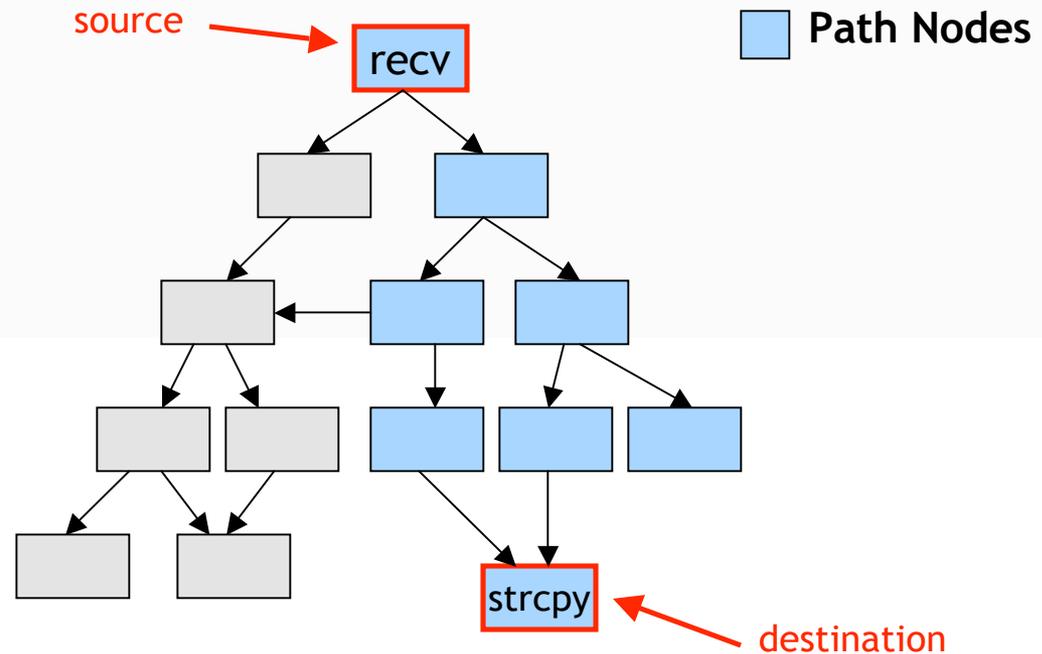




# Illustration

## Extracting The Program CFG (2)

- Identify all nodes on a path between source and target nodes

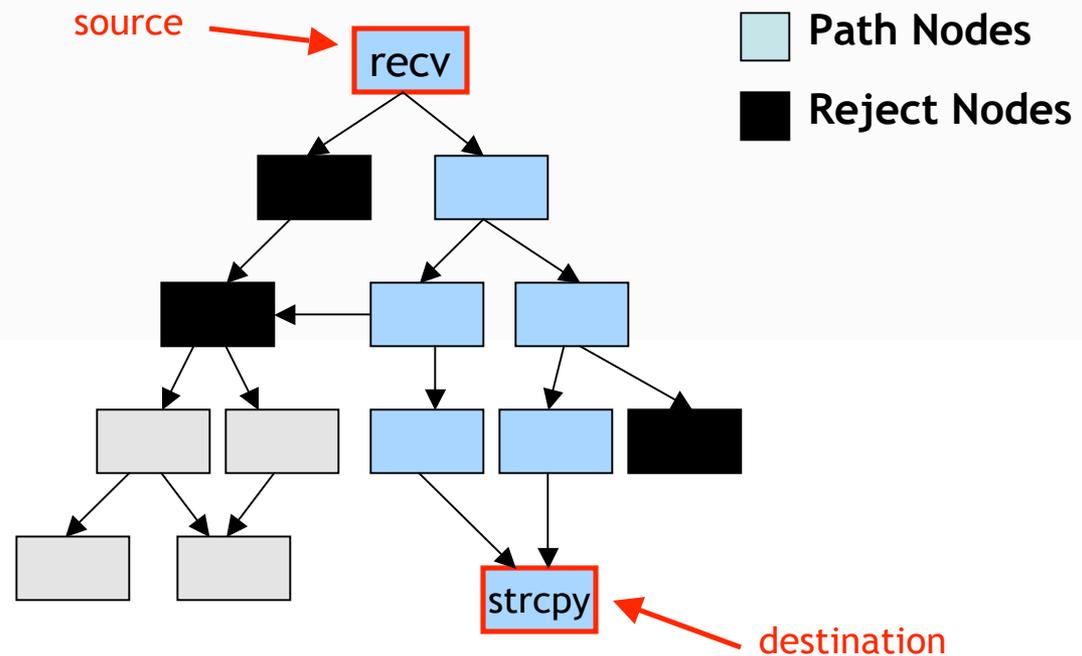




# Illustration

## Extracting The Program CFG (3)

- 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







# 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.





**DEMO**



# 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 ;)

