Software Foundations of Security & Privacy
15315 Spring 2018
Lecture 19:
Penetration Testing

Matt Fredrikson
mfredrik@cs.cmu.edu

April 12, 2018

Today's lecture

- Why are we discussing this topic?

Today's lecture

- Why are we discussing this topic?
- Final lab overview and expectations

Today's lecture

- Why are we discussing this topic?
- Final lab overview and expectations
- Penetration testing software apps
    1. Fuzz testing
    2. Concolic execution

Today's lecture

- Why are we discussing this topic?
- Final lab overview and expectations
- Penetration testing software apps
  1. Fuzz testing
  2. Concolic execution
- If time, tutorial on AFL and KLEE

Final lab

Find and exploit bugs in others' code

Final lab

Find and exploit bugs in others' code

- Given three implementations of lab0-lab2
- Find vulnerabilities that lead to security issues
- For full credit: turn in at least three bugs

Final lab

Find and exploit bugs in others' code

- Given three implementations of lab0-lab2
- Find vulnerabilities that lead to security issues
- For full credit: turn in at least three bugs

Justify the security concern

- Explain what security goal is violated
- Give proof-of-concept (PoC) exploit

Final lab

Find and exploit bugs in others' code

- Given three implementations of lab0-lab2
- Find vulnerabilities that lead to security issues
- For full credit: turn in at least three bugs

Justify the security concern

- Explain what security goal is violated
- Give proof-of-concept (PoC) exploit

Explain how to fix it

- Don't need to implement a fix
- Detailed account, implementable with minor additional effort

What's a security bug?

Zero points for correctness/availability bugs

- A crash is not enough!
- Not relevant to the goals of previous labs

What’s a security bug?

Zero points for correctness/availability bugs

- A crash is not enough!
- Not relevant to the goals of previous labs

Focus on things covered in class

- Safety: memory and control-flow
- Confidentiality: Lab 2 policies, filesystem
- Integrity: Host system should be unaffected

How should you find them?

Short answer: your choice

How should you find them?

Short answer: your choice

You are free to use whatever tools you like

- Be creative, explore the landscape
- But don't waste too much time on one tool

How should you find them?

Short answer: your choice

You are free to use whatever tools you like

- Be creative, explore the landscape
- But don't waste too much time on one tool

We have covered (or will) several in class

- CBMC
- `afl-fuzz`
- KLEE
- PIN

What to report

For each vulnerability, submit a detailed report

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!
3. Show how you exploited it, give evidence that you did

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!
3. Show how you exploited it, give evidence that you did
4. Explain in detail how to fix it ("fix the parser" is not enough!)

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!
3. Show how you exploited it, give evidence that you did
4. Explain in detail how to fix it (“fix the parser” is not enough!)
5. Provide any code that you wrote

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!
3. Show how you exploited it, give evidence that you did
4. Explain in detail how to fix it (“fix the parser” is not enough!)
5. Provide any code that you wrote

Each report should be approx. 2-3 pages

What to report

For each vulnerability, submit a detailed report

1. Identify and justify nature of vulnerability
2. Explain how you found it—we should be able to reproduce!
3. Show how you exploited it, give evidence that you did
4. Explain in detail how to fix it (“fix the parser” is not enough!)
5. Provide any code that you wrote

Each report should be approx. 2-3 pages

Key focus: we need to reproduce your findings from the report!

Grading

Each report is worth 33 points

- 15 points. Reproducible vulnerability
- 10 points. Correct fix given with adequate detail
- 8 points. Clear explanation of security issue

Grading

Each report is worth 33 points

15 points. Reproducible vulnerability

10 points. Correct fix given with adequate detail

8 points. Clear explanation of security issue

Extra credit

- Implement vulnerability fixes for your server
- Depending on scope/difficulty, 5-10 points
- Earn back missed points from previous labs!

No bugs?

What if you *really* can’t find a bug?

No bugs?

What if you *really* can't find a bug?

Still potential for full points; report on:

No bugs?

What if you *really* can't find a bug?

Still potential for full points; report on:

- Detailed steps you took to search

No bugs?

What if you *really* can't find a bug?

Still potential for full points; report on:

- Detailed steps you took to search
- What tools you used, and why

No bugs?

What if you *really* can't find a bug?

Still potential for full points; report on:

- Detailed steps you took to search
- What tools you used, and why
- Justification for your conclusion

No bugs?

What if you *really* can't find a bug?

Still potential for full points; report on:

- Detailed steps you took to search
- What tools you used, and why
- Justification for your conclusion

Convince us that there's nothing to exploit

Hacking is bug-finding

successful attack = bug-finding + exploitation

Hacking is bug-finding

successful attack = bug-finding + exploitation

How does one find bugs?

Hacking is bug-finding

successful attack = bug-finding + exploitation

How does one find bugs?

- Manually inspecting source code, reasoning about correctness

Hacking is bug-finding

successful attack = bug-finding + exploitation

How does one find bugs?

- Manually inspecting source code, reasoning about correctness
- Attempting (and failing) verification

Hacking is bug-finding

successful attack = bug-finding + exploitation

How does one find bugs?

- Manually inspecting source code, reasoning about correctness
- Attempting (and failing) verification
- Testing

Hacking is bug-finding

successful attack = bug-finding + exploitation

How does one find bugs?

- Manually inspecting source code, reasoning about correctness
- Attempting (and failing) verification
- Testing

Advanced testing techniques are widely used in security research

- Mandated in Microsoft's development lifecycle
- E.g. fuzzing uncovers "million dollar bugs" in real systems

Systematic testing

Want to be systematic in how we go about testing

This requires answers to the following questions:

- Which inputs do we choose?
- How do we check the outputs?
- When do we stop?

Partitioning: which inputs to choose

Can't test all inputs, random testing doesn't work (it's too random)

We want to find a set of tests that:

1. is small enough to run
2. is likely to catch most of the bugs we care about

Partitioning: which inputs to choose

Can’t test all inputs, random testing doesn’t work (it’s too random)

We want to find a set of tests that:

1. is small enough to run
2. is likely to catch most of the bugs we care about

Intuition: input space is very large, the program is limited

- program behavior must be “similar” on many inputs
- identify ones yielding similar behavior, pick a representative test
- make sure each input partition is covered by a test

Identifying good partitions

Partitions should correspond to relevant program properties

- Good test suite explores most of this space

Identifying good partitions

Partitions should correspond to relevant program properties

- Good test suite explores most of this space

Two basic approaches: **black-box** and **white-box**

- Black-box: as the name suggests, view the program as an opaque function and test to the specification
- White-box: use knowledge of implementation & code to generate representative tests and coverage metrics

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't miss any possible inputs

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't miss any possible inputs

Test boundary/extremal values

- Choose values close to low and high-end of valid range
- e.g., integer range, buffer size, …
- Good exercise to find holes in the specification

Black-box selection strategies

Enumerate "paths" through the specification

- Use `requires`, `ensures`, failure/exception cases
- Test each valid combination to cover all intended cases
- Also: make sure the spec doesn't miss any possible inputs

Test boundary/extremal values

- Choose values close to low and high-end of valid range
- e.g., integer range, buffer size, ...
- Good exercise to find holes in the specification

Off-nominal values

- Identify invalid inputs, choose values that test each one
- Break invariants and violate assumptions

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)
- Existence of duplicate values

Example

What are the relevant features of the `maximum` function?

```
(* if l is non-empty, returns the greatest element
   if l is empty, returns None *)
let maximum (l : int list) : int option =
  ...
```

- The size of the list (0, 1, 2, large, very large)
- Position of maximum value (beginning, middle, end)
- Range of values (negative, positive, max/min values)
- Existence of duplicate values
- Ordering of elements (ascending, descending, “random”)

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when you've covered the implementation

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when you've covered the implementation
3. Can oftentimes be automated

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when you've covered the implementation
3. Can oftentimes be automated

Disadvantages:

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when you've covered the implementation
3. Can oftentimes be automated

Disadvantages:

1. Expensive, and still not verification

White-box testing

Use details of the implementation to design and evaluate tests

- Develop partitions to maximize *code coverage*
- Test internal features like caching, domain-splitting, etc.

Whitebox testing has its own advantages

1. Tests are tailored to the code, easier to find certain bugs
2. Possible to know when you've covered the implementation
3. Can oftentimes be automated

Disadvantages:

1. Expensive, and still not verification
2. Automated tools are language-dependent, rely on heuristics

Coverage metrics: when to stop testing

Goal is to make sure tests cover all the relevant code

Coverage metrics: when to stop testing

Goal is to make sure tests cover all the relevant code

There are several ways to measure this

Coverage metrics: when to stop testing

Goal is to make sure tests cover all the relevant code

There are several ways to measure this

- Statements
- Branches
- Paths
- Traces
- ...

Coverage metrics: when to stop testing

Goal is to make sure tests cover all the relevant code

There are several ways to measure this

- Statements
- Branches
- Paths
- Traces
- ...

Each offers a different tradeoff between cost and completeness

Coverage criteria: statements

Goal: Design a test set such that each atomic command is executed at least once

Coverage criteria: statements

Goal: Design a test set such that each atomic command is executed at least once

An atomic command contains no nested statements

- Assignments, function calls are examples of primitive statements
- Loops, conditionals are not atomic

Coverage criteria: statements

Goal: Design a test set such that each atomic command is executed at least once

An atomic command contains no nested statements

- Assignments, function calls are examples of primitive statements
- Loops, conditionals are not atomic

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

no

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?

no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?

yes

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?
yes

$(n = 101, c = 2)$?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: statements

What test set achieves statement coverage?

$(n = 101, c = 1)$?
no

$(n = 101, c = 1)$,
$(n = 100, c = 1)$?
yes

$(n = 101, c = 2)$?
yes

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each branch is executed at least once

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each branch is executed at least once

Branching comes from several constructs:

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

Goal: Design a test set such that each branch is executed at least once

Branching comes from several constructs:

- conditional (if-then-else)
- `match/case`
- loops

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

What tests give us branch coverage?

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: branches

What tests give us branch coverage?

Same as before:
$(n = 101, c = 1)$, $(n = 100, c = 1)$
$(n = 101, c = 2)$

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

```
if c1 then
  if c2 then
    f1();
  else
    f2();
if c3 then
  f3();
if c4 then
  f4();
```

How many paths are in this program?

Coverage criteria: paths

Goal: Design a test set such that each *path* is executed

A path is a sequence of statements in the program:

- that takes it from an entry point to termination
- and follows the control-flow structure

```
if c1 then
  if c2 then
    f1();
  else
    f2();
if c3 then
  f3();
if c4 then
  f4();
```

How many paths are in this program?

12: 2^4 - $\{$duplicates from `c1` = $0\}$

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared
- This “bound” isn’t any better than exhaustive testing

Coverage criteria: paths

```
let f (n: int ref) (c: int ref) =
  while !c <> 0 do
    if !n > 100 then (
      n := !n - 10;
      c := !c - 1;
    ) else (
      n := !n + 11;
      c := !c + 1;
    );
  done;
  !n
```

How many paths are in this program?

Too many to test

- Bounded by width of machine integer, squared
- This “bound” isn’t any better than exhaustive testing

Loops & recursion make exhaustive path coverage infeasible

Fuzz testing

Fuzz testing

> Sitting in my apartment in Madison in the Fall of 1988, there was a wild midwest thunderstorm pouring rain and lighting up the late night sky. That night, I was logged on to the Unix system in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running ... What did surprise me was the fact that the noise seemed to be causing programs to crash.
>
> — Prof. Bart Miller

Fuzzing: what it's good for

Simple idea: feed random inputs to the program, look for crashes/exceptions

- Works in blackbox, whitebox settings
- Can be mostly random, or heavily influenced by existing tests or program internals
- In either case, it's automated: lots of inputs, no regard for norms

Fuzzing: what it's good for

Simple idea: feed random inputs to the program, look for crashes/exceptions

- Works in blackbox, whitebox settings
- Can be mostly random, or heavily influenced by existing tests or program internals
- In either case, it's automated: lots of inputs, no regard for norms

Why is this an effective technique?

- Random processes make assumptions, have biases
- Faults are good starting points for exploits
- It works: Miller found bugs in 33% of Unix utils

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Fuzzers measure coverage as they go

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Fuzzers measure coverage as they go

- Most fuzzers instrument the target program

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Fuzzers measure coverage as they go

- Most fuzzers instrument the target program
- Insert bookkeeping to count which instructions visited

Black-Box mechanics

Inputs come from many sources

- Files, standard input, network, signals, devices, ...

Common strategy: intercept syscalls

- `open`, `read`, `send`, `ioctl` ...

Fuzzers measure coverage as they go

- Most fuzzers instrument the target program
- Insert bookkeeping to count which instructions visited
- Best to rely on compiler for this, but can work on binaries

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Easy to use, often finds serious bugs

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Easy to use, often finds serious bugs
2. Test seeds can guide search towards less-random inputs

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Easy to use, often finds serious bugs
2. Test seeds can guide search towards less-random inputs
3. Seeds may also bias towards assumptions

Black-Box fuzzing: perturbation-based

Given a program and an existing test:

- Perturb the test in various ways
- E.g., flip bits, delete/append data
- See if the program crashes on any values

This approach can be extended with heuristics.

- Which tests to use as seeds?
- Strategies for perturbing seeds, ignoring certain types of input

What are the strengths and weaknesses?

1. Easy to use, often finds serious bugs
2. Test seeds can guide search towards less-random inputs
3. Seeds may also bias towards assumptions
4. Doesn't work well with checksums, grammars/protocols

Black-Box fuzzing: generational

Given a program and a format description:

1. Use the format to generate valid inputs
2. Iteratively perturb each location in the format
3. See if the program crashes on any values
4. Promote promising inputs to next generation

Black-Box fuzzing: generational

Given a program and a format description:

1. Use the format to generate valid inputs
2. Iteratively perturb each location in the format
3. See if the program crashes on any values
4. Promote promising inputs to next generation

This is a smarter way of inserting randomness

- Adhering mostly to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

Black-Box fuzzing: generational

Given a program and a format description:

1. Use the format to generate valid inputs
2. Iteratively perturb each location in the format
3. See if the program crashes on any values
4. Promote promising inputs to next generation

This is a smarter way of inserting randomness

- Adhering mostly to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

Black-Box fuzzing: generational

Given a program and a format description:

1. Use the format to generate valid inputs
2. Iteratively perturb each location in the format
3. See if the program crashes on any values
4. Promote promising inputs to next generation

This is a smarter way of inserting randomness

- Adhering mostly to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

1. Need to provide info about format

Black-Box fuzzing: generational

Given a program and a format description:

1. Use the format to generate valid inputs
2. Iteratively perturb each location in the format
3. See if the program crashes on any values
4. Promote promising inputs to next generation

This is a smarter way of inserting randomness

- Adhering mostly to the format ensures that early consistency/syntax checks are passed
- Easier to achieve coverage, requires fewer test cases

What are the weaknesses?

1. Need to provide info about format
2. Format might not match the code, lead to missed bugs

White-box fuzzing

Problem statement

Given a program and a set of inputs, generate a test set that maximizes code coverage.

White-box fuzzing

Problem statement

Given a program and a set of inputs, generate a test set that maximizes code coverage.

Main idea: Use the code itself to generate random inputs

1. Generate constraints that reflect the program's control flow
2. Solve the constraints, map solution to corresponding inputs
3. Run program on these inputs, look for crashes or exceptions

This idea was pioneered by Patrice Godefroid at Microsoft

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

$$x = \texttt{SHA1}(\ldots) \wedge y > 3 \wedge x < y \wedge \ldots$$

Static test generation

This isn't always possible

```
if x = SHA1(...) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Second path: execute `f1, f3`

$$x = \texttt{SHA1}(\ldots) \wedge y > 3 \wedge x < y \wedge \ldots$$

Solving requires finding `SHA` pre-image

Dynamic symbolic test generation

Generational testing++

Dynamic symbolic test generation

Generational testing++

Given a program and test case:

1. Run the test case, and collect constraints along the tested path
2. Modify constraints by negating selected *literals*
3. Solve new constraints, generate corresponding inputs
4. Repeat until all assertions are reached [Korel 1990, ...]
5. Or, generate inputs for all feasible paths [Godefroid et al 2005]

Dynamic symbolic test generation

Generational testing++

Given a program and test case:

1. Run the test case, and collect constraints along the tested path
2. Modify constraints by negating selected *literals*
3. Solve new constraints, generate corresponding inputs
4. Repeat until all assertions are reached [Korel 1990, ...]
5. Or, generate inputs for all feasible paths [Godefroid et al 2005]

This approach is called DART (**D**irected **A**utomated **R**andom **T**esting)

Example

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$\texttt{assume}(x = \texttt{SHA1}(z))$
$\texttt{assume}(y > 3)$
$\texttt{f1}();$
$\texttt{assume}(\neg(x < y))$
$\texttt{assume}(y > 3 \;\&\&\; x \geq y)$
$\texttt{f4}()$

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$\texttt{assume}(x = \texttt{SHA1}(z))$
$\texttt{assume}(y > 3)$
$\texttt{f1}();$
$\texttt{assume}(\neg(x < y))$
$\texttt{assume}(y > 3 \,\&\&\, x \geq y)$
$\texttt{f4}()$

We can still explore `f2, f3` by changing y

We fix x and z, change other literals

Example

```
if x = SHA1(z) then
  if y > 3 then
    f1();
  else
    f2();
if x < y then
  f3();
if y > 3 && x >= y then
  f4();
```

Start with `x = 5, y = 4, z = 0`

Assume that $\texttt{SHA1}(0) = 5$

This yields the path:

$$\texttt{assume}(x = \texttt{SHA1}(z))$$
$$\texttt{assume}(y > 3)$$
$$\texttt{f1}();$$
$$\texttt{assume}(\neg(x < y))$$
$$\texttt{assume}(y > 3 \text{ \&\& } x \geq y)$$
$$\texttt{f4}()$$

We can still explore `f2, f3` by changing y

We fix x and z, change other literals

$$x = 5 \wedge z = 0 \wedge \neg(y > 3) \wedge \neg(x < y) \wedge \ldots$$

More intelligent search

Start with a well-formed seed test

More intelligent search

Start with a well-formed seed test

Generate the path constraint

- Negate each literal independently
- Generate a new test for each negation, add to test set
- Repeat until resources run out, or we have path coverage

More intelligent search

Start with a well-formed seed test

Generate the path constraint

- Negate each literal independently
- Generate a new test for each negation, add to test set
- Repeat until resources run out, or we have path coverage

This approach tests many “layers” of the program early

Contrast with classic depth-first approach

Static test generation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 4) crash();
}
```

input = "good"

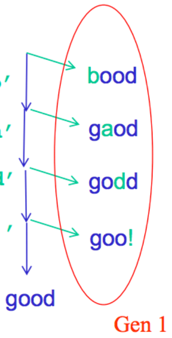
Path constraint:

I_0!='b' → I_0='b'

I_1!='a' → I_1='a'

I_2!='d' → I_2='d'

I_3!='!' → I_3='!'



Negate each constraint in path constraint

Solve new constraint → new input

Example from Patrice Godefroid

DART implementations

This approach has been used in many tools

- EXE (Stanford), concurrently with Godefroid's original work
- CUTE (Bell Labs), concurrently with original work
- SAGE (Microsoft Research)
- PEX (Microsoft Research)
- YOGI (Microsoft Research)
- Vigilante (Microsoft Research)
- BitScope (CMU/Berkeley)
- CatchConv (Berkeley)
- Splat (UCLA)
- Apollo (MIT/IBM)