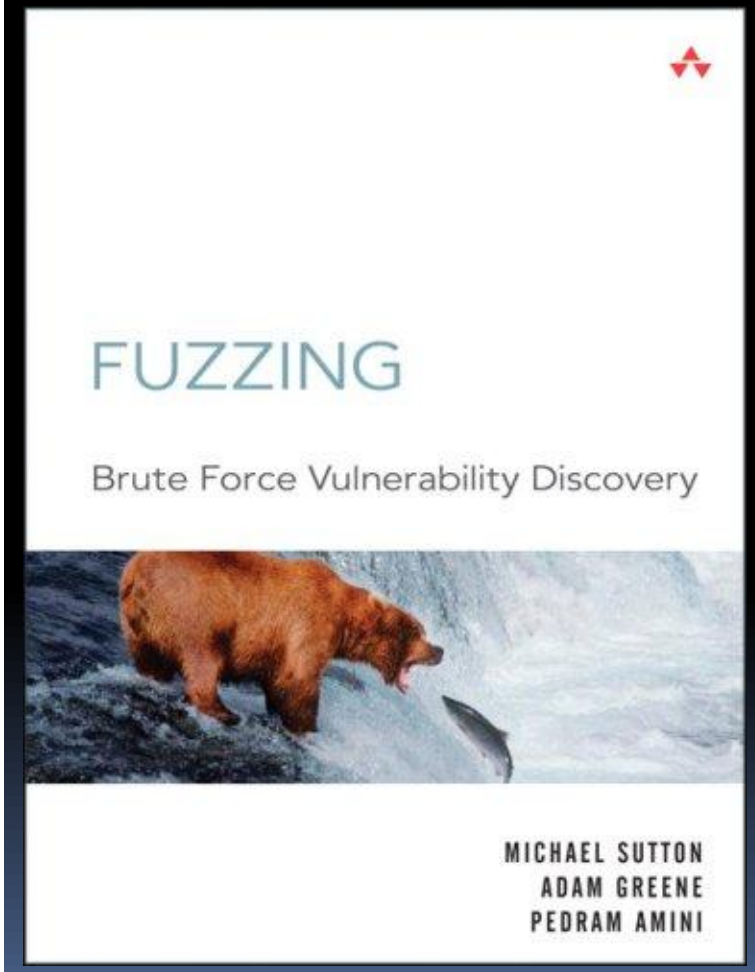
Adam Comella
Jay Smith

FUZZING

Resources



http://www.fuzzing.org/

Security auditing methods

- Source code analysis
  - RATS, Jlint, etc.
- Binary analysis
  - Static
    - IDA Pro, Bug Scam, etc.
  - Dynamic
    - Debugging, hit tracing, fuzzing

Security auditing categories

- Whitebox
  - Source code is available
- Graybox
  - Only compiled binary available
- Blackbox
  - Control over input
  - Output can be observed

What is fuzzing?

- Fuzzing is the process of automatically feeding data to a program with the intent of causing the program to crash or expose a bug
- Data can be
  - Random data
  - Pre-generated test cases
  - Legitimate input data that has been mutated
  - “Smart” data generated by a grammar

WTFuzz

- Fuzzing can be traced back to the University of Wisconsin in 1988
  - Professor Barton Miller's "Operating System Utility Program Reliability – The Fuzz Generator" assignment
- 1999 – Oulu University starts PROTOS
- 2002 – Dave Aitel's SPIKE
- 2004 – Mangleme by Michael Zalewski
- 2005 – FileFuzz, SPIKEfile, Codenomicon
- 2006 – ActiveX fuzzers COMRaider and AxMan

Fuzzing targets

- File formats
- Network protocols
- Command-line args
- Environment variables
- Web apps

Getting your feet wet

- Simply piping /dev/urandom to a target program is one method of fuzz testing

Getting your feet wet

- Simply piping /dev/urandom to a target program is one method of fuzz testing
  - How effective is this?

Getting your feet wet

- Simply piping /dev/urandom to a target program is one method of fuzz testing
  - How effective is this?
- Another simple fuzz testing method is hooking `getenv()`

Getting your feet wet

- Simply piping /dev/urandom to a target program is one method of fuzz testing
  - How effective is this?
- Another simple fuzz testing method is hooking `getenv()`
  - How about this method?

Other local fuzzing targets

- `argv[0]` is sometimes trusted too much
- Command line args can also be fuzzed
  - iFuzz command line fuzzer
  - Usage output can be analyzed to aid this type of fuzzing
  - If the program uses `getopt()` then more info can be leveraged

Sulley – A fuzzing framework

- Named after this fuzzy guy →

Sulley - A Fuzzing Framework

- Named after this fuzzy guy →

Sulley - A Fuzzing Framework

- Provides an environment for:
  - Pregaming
    - Describing data and protocols
  - Fuzzing
    - Mutating data
    - Logging crashes and all data generated
    - Restarting the target when it crashes
  - Postmortem
    - Investigating the cause of a crash

Describing Data

- Data is described by a sequence of Python functions
  - “hello, 24” described by:
  - s_string(“hello”)
  - s_delim(“,”)
  - s_long(24)
- Each of the above 3 fields get mutated during fuzzing

Primitives

- The basic foundations of your data
- Integers
  - s_char(), s_short(), s_long(), s_double()
- Strings
  - s_string()
- Static values
  - s_static(), s_binary()
- Misc
  - s_delim(), s_random()

Primitives - Integers

- Functions
  - s_char(), s_short(), s_long(), s_double()
- Required parameters
  - default value - s_short(24)
- Other options: endianness, signed, use full range when fuzzing
- Mutations
  - smallest values in the range (0, 1, etc.)
  - largest values in the range (254, 255, etc. for char)

Primitives - Strings

- Functions
  - s_string()
- Required parameters
  - default value - s_short("hello, world")
- Other options: length, pad character
- Mutations (strings that cause problems)
  - a variety of long strings (AAAAA…)
  - format strings (%n%n%n…)
  - empty string ("")

Primitives - Static

- Never mutate during fuzzing
- s_static()
  - Takes a string: s_static(“HTTP”)
- s_binary()
  - Takes input in a variety of hexadecimal formats
  - s_binary("oxde be ef \xca fe 00 01 oxbaoxdd fo od")
- Mutations
  - None

Primitives - Delimiters

- s_delim()
- Required parameters
  - original delimiter: s_delim(":")
- Mutations
  - omitted delimiter ("")
  - repeated delimiter (":::::::::")
  - other common delimiters ("!", "=", ";")

Primitives - Random

- s_random()
- Generates a random chunk of data of a certain length
- Required parameters
  - initial value
  - minimum length
  - maximum length
- example: s_random("GET", 10, 15)

A Problem

- In describing a protocol, what if we need to include
  - the length of a string?
  - the checksum of a section of data?
- Our data is constantly being mutated so how can we possibly include these values?
- This is what blocks are for!

A Solution - Blocks

- Give a name to a section of data
- To include a size or checksum in your data refer to the data block by name
- s_block_start(name_of_block)
- s_block_end()

Block Helpers - Sizers

- s_size(block_name)
- Include the size of a block in your data
- Other options
  - how many bytes is the size field?
  - endianness
  - include length of size field in size?
  - fuzz this parameter? (default is NO)

Block Helpers - Checksums

- s_checksum(block_name)
- Include the checksum of a block in your data
- Other options
  - algorithm (crc32, adler32, md5, sha1, custom)
  - endianness
  - checksum length

Block Helpers - Repeaters

- s_repeat(block_name, min_reps, max_reps)
- Repeat a block a variable number of times
- Other options
  - step - how much should reps be incremented for each fuzz?

Block Helpers - Example

- Protocol
  - types: [byte][string][short][crc32]
  - values: [length-of-name][user-name][health][cksm]
- if s_block_start("packet"):
  - s_size("user_name")
  - if s_block_start("user_name"):
    - s_string("a user name")
  - s_block_end()
  - s_short(55)
- s_block_end()
- s_checksum("packet")

Groups

- Specify a list of static values
- Attach group to a block: the block will cycle through the values of the group as a prefix
- Useful for representing verbs and opcodes

Groups - HTTP Request Example

- s_group("http_verbs", ["GET", "POST", "HEAD"])
- if s_block_start("body", group="http_verbs"):
  - s_delim(" ") s_delim("/") s_string("index.html")
  - s_delim(" ") s_string("HTTP") s_delim("/")
  - s_string("1") s_delim(".") s_string("1")
  - s_static("\r\n\r\n")
  - s_block_end()
- example outputs:
  - GET /index.html HTTP/1.1
  - POST /index.html HTTP/1.1

Requests

- Primitives -> Blocks -> **Requests**
- Recall primitives are the simplest unit for describing data
- A request
  - is built up from blocks and primitives
  - generally describes a complete conversation you may have with a target
- When fuzzing you will tell Sulley "fuzz this request on this target"

Requests - Syntax

- s_initialize(request_name)
  - Creates request_name
  - Makes request_name the current request
- When primitives and blocks are described, they are added to the current request
- Requests are terminated by the next call to s_initialize()
- Last request is unterminated

Monitoring while Fuzzing

- Process Monitor
  - Logs crashes (we're fuzzing to find crashes)
  - Restarts target when it crashes (so we can keep fuzzing without human intervention)
- Network Monitor
  - Logs all network traffic associated with your fuzz (makes it easier to reproduce & understand crashes)

Monitoring while Fuzzing

- Virtual Machine Monitor
  - Useful when running the target in a VM
  - Start & stop VM
  - Restore VM to stable snapshot

Drivers - Bringing it All Together

- In the driver, you:
  - Select the target
  - Setup the monitors
    - process monitor, network monitor, VM monitor
  - Select the requests to fuzz
  - Fuzz!

- Let's look at simple_driver.py

Postmortem

- We have some crashes that we must investigate!
- crashbin_explorer.py
  - Lists crashes from a fuzz
  - Investigate stack and register states at time of crash

Postmortem - Isolating Malicious Data

- Crash may happen after 100th test case
  - sending all 100 test cases to play with crash is too much!
- Try sending just the 100th test case but it may not cause a crash
  - need an earlier test case to put target into vulnerable state

When to fuzz

When to fuzz

- During a 8 hour security competition?

When to fuzz

- During a 8 hour security competition?
- During a 48 hour security audit?

When to fuzz

- During a 8 hour security competition?
- During a 48 hour security audit?
- Hired to do QA on a piece of software?

When to fuzz

- During a 8 hour security competition?
- During a 48 hour security audit?
- Hired to do QA on a piece of software?
- In your spare time?

404 Bug not found

- Misconfiguration bugs
- Design flaws

Take away

- A good fuzzer should
  - Have a flexible way to describe a protocol or format
  - Log all test cases
  - Monitor the target for signs of a bug
  - Correlate test cases to crashes

Lab on Monday

- Using Sulley to fuzz a protocol
- Sulley works on Windows and Mac OS X