The Evolving Art of Fuzzing

DEFCON 14

August 5th 2006

Jared DeMott

Vulnerability Researcher - Applied Security, Inc.

(jdemott@appliedsec.com, www.appliedsec.com)

APPLIED SECURITY™

Dr. Richard Enbody

Associate Professor - Michigan State University

(enbody@cse.msu.edu, www.cse.msu.edu/~enbody/)

MICHIGAN STATE UNIVERSITY

Agenda

1. Definitions and Motivation
   - Who/what/where/why/when
2. Current state of fuzzing
   - The various types of fuzzers and how they work
3. Kung FU with a context-free grammar fuzzer
   - Widely used and sold - Codenomicon
4. Serious Kung FU with a generic fuzzer
   - See GPF or Autodafe
5. Web Fuzzing
   - SPI Fuzzer
6. Fuzzing Metrics
   - Formal study of fuzzing
7. Advancing the state of fuzzing
   - Adding to generic fuzzers, and Genetic Algorithms

Background

- Section I
  - Define Fuzzing
  - Define Software Testing
  - Figure out how the two fit together
  - Understand the usefulness of fuzzing
    - A practical automated test tool that finds bugs

Definition

- ***"Fuzzing — a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. From modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines." - Oehlert***
  - ***See provided paper for other useful fuzzing terms***

Fake Clear Text Protocol

[Client]-> “user jared\r\n”
“user OK. Provide pass.\r\n” <-[Server]
[Client]-> “pass mylamepasswd\r\n”
“Login successful. Proceed.\r\n” <-[Server]
[Client]-> “list file 1\r\n”
...

Simple Fuzz Example

Consider a fuzzer that randomizes the test type, position, and protocol leg in which to place the attack:

[Client]-> “us<*50000 \xff’s*>er jared\r\n”

----------loop 1----------

[Client]-> “user ja<*12 %n’s*>red\r\n”

“user Ok. Provide pass.\r\n” <-[Server]

[Client]-> “*\x34\x56\x12\x…*\r\n”

----------loop 2----------

[Client]-> “user ja<*1342 \x00’s*>red\r\n”

----------loop 3----------

[Client]-> “user jared\r\n”

“user Ok. Provide pass.\r\n” <-[Server]

[Client]-> “*\x04\x98\xbb\x…*\r\n”

----------loop 4----------

Other Terms you may have Heard

- ***Monkey, Stochastic, boundary or stress testing are all a little different (or not) depending on who you talk to. Some have even used the term fault injection to mean fuzzing, but typically that means something else. For uniformity fuzzing is the term we will use.***

Software Testing

- Software testing can be
  - Difficult, tedious, and labour intensive
  - Poorly integrated into the development process
  - Abused and/or misunderstood
- Software testing is expensive and time-consuming
  - Typically at least 50% of initial development costs
- Primary/only method for gaining confidence in the correctness of software (pre-release)
- In Short, testing is a hard problem

Software Testing

- Functional Testing (Dynamic) (Black-box)
  - Executes the software
  - Tends to focus on final requirements, system stability, and exposed interfaces
    - *Pro: Real code compiled for real environment*
    - Con: Complexity of search space (infinite). Poor test case creation (could test the same path over and over)
- Structural Testing (Static) (White-box)
  - Symbolically execute software
  - Tends to focus on design and code correctness
    - *Pro: Can be done early in the unit phase. Once found, problems are easier to troubleshoot. Can test code.*
    - Con: Requires source code. Manual reviews are difficult.
- For both, commercial tools are expensive

Where does Fuzzing fit into Testing?

- ***This question will be answered differently by each company. But in general we have:***
  - ***Formal Methods in Software Engineering***
    - ***Software Quality Assurance***
      - ***Software testing***
        - ***QA people and SR people could learn from each other***
- ***Fuzzing is one of many software testing techniques***
  - ***Many other types of testing***
    - Grey-box is a combination common to security testing
    - Other testing types include: Unit, integration, system, end-to-end, performance, usability, functional, load

Is Fuzzing "better"?

- Do fuzzers replace source code audits, reverse engineering, or other software quality assurance processes?
  - No. Fuzzing compliments, supplements, or helps complete all those activities, but it does not replace anything.
  - No one software testing technique will ever have the final word on overall software quality.

So, how good are Fuzzers?

- ***From Barton Miller in 1990 to Martin Vuagnoux in 2006 (16 years), fuzzers have performed surprisingly well. Fuzzers have traditionally been "quick and dirty" without much formal study.***
  - ***Some have begun to change that: Miller, Aitel, PROTOS folks, Ohelert, Sprundel, Sutton/Greene, Vuagnoux, commercial companies, and many more.***
    - ***We too are studying fuzzers more formally***

Who builds Fuzzers?

- Software companies
  - Proactive security testing
- Vulnerability analysts
  - Security research
  - Income
- Fuzzer companies
  - Income
- Academia
  - Advancements in the field
- Hax0rs
  - Who knows why -- phun, profit, hobby, or because they can! :)

Why are Fuzzers built?

- Because they find bugs
  - They are built to find bugs in a way that is different from traditional testing methods. Software testing has been part of computing since the inception of computers and has been researched intensively, but still bugs persist.

How Fuzzers Work

- Section II
  - Various types of fuzzers and how they create semi-valid data

What do Fuzzers do?

- Fuzzers deliver semi-valid data to the target (software under test) and optionally determine if a fault has occurred.
  - Automated tool that functionally tests a program
  - The source of this data and how it becomes semi-invalid is important
    - Attack heuristics such as integer and string **bounds checking**, format characters, out of order commands, bad delimiters or line endings, etc.
  - attack surface
  - fuzz(source)->attack surface<-debugger

Why do Fuzzers work?

- A general goal to break software
  - Traditional testing focuses on proper functionality, not security testing. Errors of omission are an interesting example. (bounds check)
- Code Coverage
  - A false sense of security. Coverage tells us something, but not the complete story.
- Gap Coverage
  - Researcher's testing tools/techniques different from creators
- Intelligent randomness
  - All paths + all data == infinite problem

Fuzzer Types

- Generation
  - Full internal description of protocol, one-for-one, less total but special tests, could achieve better coverage, possibly new effort for each protocol
- Mutation
  - Capture file (can be modified), generic, more up front effort but rapid fuzzing of almost any PT protocol, heuristics expand each project
- Fuzzing frameworks and fuzzer scripts
  - spike, peach, etc. Facilitate the rapid creation of block based fuzzers.
- Pure random stream generators
  - Old school, but have still found bugs

Creating semi-valid data

- Test Cases
  - Tools for sale
- Cyclic
  - Deterministic runs
    - 1 to 10000 bytes inserted in each position on each line/leg incremented by 1 byte (0x00-0xff)
- Random
  - Infinite runtime
    - with intelligence could cover more of the input space in a finite time
- Library
  - List of attack heuristics tried on each “variable”
- Combination of some or all (GPF)

What is intelligent fuzzing?

- Notion of randomness (dumbness) and protocol specific knowledge (intelligence)
  - Purely random data has found a few bugs in the past but will likely get dropped really fast really often
  - Too much intelligence can be expensive
    - Could also lead to some of the same poor assumptions coders made

Which fuzzer is best?

- No published research has been done
  - Depends on protocol/application, project, experience of testers, time, budget, available tools, etc.
  - Pros/cons
    - generation/generic, dumb/intelligent, randomness/lists, logs/debugger, etc.

Context-Free Grammar Fuzzers

- Section III
  - CFG fuzzers are one possibility for a generation fuzzer
    - Use the Oulu University Secure Programming Group's PROTOS as an example

What is a Context-Free Grammar (CFG)?

- *A formal grammar in which every production rule is of the form*
  - *"V → w, where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. The term "context-free" comes from the fact that the non-terminal V can always be replaced by w, regardless of the context in which it occurs. A formal language is context-free if there is a context-free grammar that generates it." - Wikipedia*
- *Backus-Naur Form* (BNF) is the most common notation to express context-free grammars.
  - Regular expressions is another CFG example

CFG Example

- CFGs are generation fuzzers, since a complete description of the protocol is required
  - More likely a deterministic runtime
    - Could be *infinite*
      - Add randomness to tests, and wrap with a *while(1)*
- The PROTOS and Codenomicon folks use CFG
  - See Rauli Kaksonen, “A Functional Method for Assessing Protocol Implementation Security”
  - Also
    - ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/

CFG Example

<transfer> = <read-transfer> | <write-transfer>
<read-trasfer> = !up<RRQ> <reads>
<reads> = {!down<BLOCK> !up<ACK>} !down<LAST-
BLOCK> !up<ACK>

<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512 x
<OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 {
<OCTET> }
<ACK> ::= (0x00 0x05) <BLOCK-NUMBER>

<MODE> ::= "octet" 0x00 | "netascii" 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
<OCTET> ::= 0x00 - 0xff

CFG Example

- PROTOS
  1. Round-up of interfaces, which the software uses to get input, especially interfaces to external systems.
  2. Specification of protocols used by the tested interfaces. One specification will do if multiple products implementing the same protocol are being tested.
  3. Execution of tests.
  4. Inspection and verification of test results.
- Vulnerability testing has same limitations as syntax testing
  1. The tested software may behave completely inappropriately according to specifications, even if it has passed all tests.
  2. Vulnerability testing is only likely to reveal errors in software implementation, as specification and design errors require complex test-cases with a specific sequence of events and conditions.
  3. Not everything can be monitored (this applies to all software testing). The ways to compromise security are unlimited whereas we can only monitor limited aspects of behavior

CFG Example

- PROTOS
  - Wide use (of any fixed tool) is likely to cause a pesticide-paradox: a software product which is tested will become immune to it.
    - Plan to make PROTOS more sophisticated and therefore expose more subtle vulnerabilities
    - There are always vulnerabilities not discovered
    - Baseline:
      - Products below the baseline are insecure.
      - Products above the baseline do not contain the (trivial) vulnerabilities searched by the test-tool.

Generic Fuzzers

- Section IV
  - Our current vision for a General Purpose Fuzzer
    - GPF and Autodafe do much of what I'll mention

Kung FU with a Generic Fuzzer

- Automatic Protocol Detection
  - Capture valid session
  - Convert to neutral format
    - Manual modification
  - Plug-in capable for complex protocols
    - Manipulate received data
- Tokenize
  - Strings, binary data, length fields
    - Automatically detect and associate with known attack heuristics
- Strong Attack Heuristics

Kung FU with a Generic Fuzzer

- Intelligent randomness
  - Very little research has been done on how/when to apply attack heuristics if done in a random manner
- Remote Debugging
  - Real time statistics, dynamic weighting, and fault detection
- Distributed Fuzzing
  - Fuzz/Debug Server => Fuzzers => Fuzzies
  - Could be totally random session
    - Hard to determine what happened

Web Fuzzing

- Section V
  - Why is HTTP special compared to FTP, SMTP, POP3, IMAP, etc?

Web Fuzzing

- Security testing of HTTP applications is much different that traditional network applications
  - It's not all that likely that you'll find a new Apache or IIS bug.
  - It's very common to find a file inclusion or other (PHP, ASP, etc) bug in a web application that runs on top of Apache or IIS.
  - We certainly can and should fuzz HTTP, but once that's done we need to turn to the less well known (and thus less tested) applications
  - SPI Dynamics has a useful tool called SPI Fuzzer
    - Looking at return data such as HTTP error codes

Fuzzing Metrics

- Section VI
  - All six slides loosely quote Martin Vuagnoux
    - He's the first to bound fuzzing with a meaningful complexity

Fuzzing Metrics

- Potential Space of Inputs
  - The cardinality of the potential space of inputs defines the complexity of fault injectors: fuzzers basically substitute variables for smaller, bigger and malformed strings or values. By using a random character string generator, Fuzz (by Miller) owns an infinite potential space of inputs. In order to reduce the complexity, most advanced fuzzers combine three techniques:
    - Partial description of protocols. In order to omit useless tests.
    - Block-Based protocols analysis. This technique permits to recalculate length fields after substituting data.
    - Library of substituted strings or values.

Fuzzing Metrics

- The complexity of Autodafe is L*F
  - L = number of substituted strings or values
    - Using a library of finite substituted strings or values drastically reduces the size of the potential space of inputs. E.g. in order to highlight format string bugs, only a few character strings are tested, containing all the interpreted sequences.
    - L should be “dozens of thousands”.
    - Built up as new attacks are discovered
  - F = number of fuzzed variables
    - These are the parameters (data) sent to the attack surface. In general, the RFC helps us count these. A partial set can be captured live.
    - Arranging or reducing will decrease runtime

Fuzzing Metrics

- Weighting Attacks with Markers Technique
  - Removing even one input in F is profitable
  - Use a tracer/debugger
  - Determine which variables get consumed by dangerous functions
    - printf, vprintf, vsprintf, wprintf, vwprintf, vswprintf, sprintf, swprintf, fprintf, fwprintf, getenv, strcat, strncat, strcpy, strncpy, stpcpy, memcpy, memccpy, bcopy, memmove, gets, system, popen, scanf, sscanf, fscanf, vfscanf, vsscanf, realpath, fgets, etc.
  - Fuzz such variables first
    - This orders the complexity, making the fuzzer run that much more efficient

Advancing the Art

- Section VII
  - A peek at our future research

Advancing Fuzzing

- “Substituting variables with random values is irrelevant.” - Vuagnoux
  - It will force the runtime toward the input space (infinite)
    - But is this always a bad thing? Couldn't it find flaws that libraries miss? How were buffer overflows first discovered? What about 2nd generation bugs?
    - Can we find/define a good stop point with a near infinite input space?
  - Lets use both approaches (no I'm not a politician)
    - If the list attack fails we move on to a more unbounded attack.
      - Variable number of sessions, crazy states, heavily randomized data, clustered or out of order commands, etc.
- Genetic Algorithms
  - Sounds crazy doesn't it?

Discussion Time

- Love to chat:
  - What defines a good fuzzer?
  - How long should it run?
  - How else, what else, could we fuzz?
    - Virtual OS or hardware?
  - Should fuzzer or vulnerability scanner companies blend the two technologies?
- For more info/fun with fuzzing:
  - Run cmdline, ikefuzz, and GPF on the DEFCON CD
    - All three have found bugs
  - Read fuzzing paper (also on CD)
  - Stay tuned...we're going to keep moving forward with fuzzing research! :)