A Maze of Twisty Passages all Alike: A Bottom-Up Exploration of Open Source Fuzzing Tools and Frameworks

Matthew Franz

mdfranz@threatmind.net   @frednecksec

CERT Vulnerability Discovery Workshop (Feb 2010)
Agenda

• Introduction
• Beyond smart & dumb fuzzers
• A Case Study in Fuzzer Selection
• Conclusions (and stuff I ran out of time on)

Source: http://www.colossalcave.com/cavetour.html
Where I’m coming from...

- Lots of “big company” security QA/R&D during early-mid 2000s
  - Primarily dealt with binary protocols on embedded devices
  - Wrote a variety of protocol-specific fuzzers and two attempts at block-based multi-protocol fuzzing frameworks (in Python/C#)
  - Used some commercial tools near the end
- Some on-the side (mostly unbillable) vuln research in a small SCADA security consulting firm
  - If Amap and Nessus find bugs, your fuzzers can be pretty crude
  - Still somewhat traumatized by the SCADA disclosure debate
- Enjoyed a sabbatical from vuln research & pen-testing from late 2006 to mid-2009, but slowly getting back into it again
  - Sneak some robustness testing in compliance engagements
  - Focusing Smart Grid (AMI), SCADA redux, etc.
  - Trying to resist the temptation of writing new tools from scratch
Fuzzing in 2010

• No longer exotic/boutique
  – Responsible for some non-trivial % of vulns discovered
  – Even integrated into commercial signature-based vuln scanners
• Over 100 fuzzers on Jeremy Brown’s list
  – Range of capabilities and usability/usefulness
  – Dormant to active development
  – Crude Perl hacks to well-defined documented APIs
• Can there be too many choices?
Objectives & Non-Objectives of this Talk

• Try to untangle the “maze” of FOSS fuzzers by:
  – Isolating the discrete feature-sets most useful for performing efficient software security testing
  – Developing a framework for evaluating and selecting tools for specific users & use cases
  – Identifying common (and useful) design & implementation approaches and highlight some standouts and areas for development

• Avoiding some more interesting problems
  – Coverage metrics
  – Effectiveness and track record of tools
  – Fuzzing bake-off vs. reference implementations
  – Commercial vs. Open Source capabilities
Who uses Fuzzers and why do we care?

• QA/test engineers
  – “Click on start” and give me a traffic light when done
  – Coverage, repeatability, test case reduction are a major concern

• Pen-testers of various shapes & sizes
  – That probably know how to do a little scripting
  – That should know how protocols work on the wire
  – A single bug might be good enough

• Hard core bug hunters
  – That could implement the protocols they are testing (in .asm)

This diversity of objectives, backgrounds, requirements, programming/scripting languages has led to the “the maze”
Exploration Approach

• Biases
  – Religious conviction that C (and Perl) should be avoided at all costs and that simple small lightweight tools are always best
  – Selfish interest in binary & proprietary network protocols
  – Which tools would be the most useful for some upcoming projects and that could be used by members of my team (who have less experience with robustness testing)

• Evaluation criteria
  – Tools had to support multiple protocols /applications/file format
  – Compiled relatively easily on a recent version of Ubuntu
  – Open Source only (wasn’t anal about license terms)
  – Web client/server tools were sufficiently different to exclude them

• Analysis process
  – Too much time reading through source code and trying to get them to work
  – Not enough time fully testing all the features on real protocols
  – Focus was on identifying discrete attributes (see the .xls for the raw data)
  – Validated scheme based on a larger number of tools and then narrowed down
BEYOND SMART & DUMB FUZZERS
Attributes of Fuzzers/Frameworks

• Target – external interface under test
  – Client, Server, Parser, Kernel, Protocol, etc.
• Mode of Operation
  – API
  – Executable
• Language – Python, C, Ruby, etc.
• Transport – you can inject test cases into the application/protocol (TCP, IP, UDP, SSL, IPv6)
• Template
  – Generation – manual automated, inline, from traces, file source
  – Data Model – representation of messages and protocol state
  – Built-in Functions – crypto, checksum, hashes, encoding, etc.
Attributes of Fuzzers & Frameworks (cont.)

• Fault Payloads
  – “canned” vs. programmatic
  – buffer overflow, format string, bit shifting, etc.

• Debugging & Instrumentation
  – Fault detection
  – Control and monitoring of target (both internal

• Session Handling
  – Capture, storage, replay
  – Logging
  – Interactive vs. Unattended
  – Pause, stop restart, breakpoints

• Documentation & Examples

See the spreadsheet for the details...
Attributes & Workflow (all features)

Target
Debugging

Capture / Conversion
Generation
Inline Proxy

Logging
Transport (injection)
Session Recording & Replay
Breakpoints

Template
Data Model
State Model

Payloads

Mode of Operation
Operating Modes

• Approaches
  – API-based
    • Write code in a scripting language
    • Extend existing processors
    • **Examples**: sulley, ruckus, peach, fuzzled
  – Executable
    • Execute fuzzing engine against a more/less complex configuration file with more/less complex command-line options
    • **Examples**: peach, GPF, autodafe

• Primary consideration: time to test/develop
  – Go with executable if you have limited time
  – If you have to partially implement the protocol anyway you should probably go with API
  – Some configurations files (templates) are more convoluted that coding
More on Templates

• Template development is the most tedious (and sometimes difficult) process of modeling the valid/invalid data

• Auto generation of an “unknown” protocol remains a “holy grail” problem
  – This is was the point of the protocol informatics (PI) project
Example Template Files

```c
block_begin("packet 3");
block_end("packet 3");
send("packet 3"); /* tcp */

block_begin("packet 4");
block_begin("packet 4.6.54.mbtcp");
  // name : modbus_tcp.trans_id
  // showname: transaction identifier: 0
  // show : 0
  // size: 0x2 (2)
  hex(
    00 00
  );
  // name : modbus_tcp.prot_id
  // showname: protocol identifier: 0
  // show : 0
  // size: 0x2 (2)
  hex(
    00 00
  );
  // name : modbus_tcp.len
  // showname: length: 6
  // show : 6
  // size: 0x2 (2)
  hex(
    00 06
  );
  // name : modbus_tcp.unit_id
  // showname: unit identifier: 1
  // show : 1
  // size: 0x1 (1)
  hex(A)
```

Autodafe (Modbus/TCP)  GPF (MongoDB)
A Peach Template

Single XML file contains message format, states, and injection commands
Auto Template Generation

• Approaches
  – PDML*
    • Autodafe - pdml2ad generates block based description based on
    • Peach – allows creation of Peach pit
  – Pcap
    • GPF – creates text file (.gpf) that is replayed (with multiple malformation options)
  – Inline
    • Taof

• Caveats
  – Best to just use a single stream
  – PDML requires a Wireshark dissector

* Not Open Source but pcapr.net does this and JSON file that you can run with muDos to inject the packets against a target
Payload Generation

• Approaches
  – Primitive randomization
    • Tcpjunk, isic, GPF pure mode
  – “CGI-Scanner”-style dictionary of known bad requests (format strings, strings and numeric input to test boundary conditions
    • 4f, autodafe, SPIKE
  – Various mutation APIs
    • Peaches, Ruckus, Antiparser
Tools by Development Status (Last Release)

**Recent Development**
- Tcpjunk (1/2010)
- Peach (1/2010)
- Sulley (2/2009)
- Ruckus (4/2009)

**Apparently Dormant**
- Fuzzled (10/2007)
- Autodafe (8/2006)
- Scratch (9/2004)
- SMUDGE (9/2004)
- GPF (Jared?)
Dealbreakers: Active Projects

- Peach
  - Robust set of features but a huge learning curve and insane dependencies (a 20MB installer?)
  - Not Linux/OSX friendly
  - PDML conversion disappeared/is hidden in 2.3.x
  - Maybe I can reuse some of the APIs

- Tcpjunk
  - No example templates
  - No way to automatically create them
  - ASCII protocol bias
Recommended Improvements for the “Keepers”

• GPF
  – Write some wrappers for command-line arguments

• Taof
  – Better representation of binary protocols and marking of “fuzz points”

• Sulley
  – Automatic generation block descriptions
A CASE STUDY IN TOOL SELECTION
Fuzzing MongoDB in 20 minutes (hypothetically)

- What is MongoDB?
  - Document oriented #nosql database (in the same family as CouchDB)
  - Written in C++ (with broad driver support in various scripting languages)
  - Uses SpiderMonkey (or Google V8) for its .js engine – queries are in JavaScript (and JSON)
  - Has a proprietary JSON like serialization protocol called BSON

Selecting your fuzzer: info gathering

• Do you have a protocol specification?
• Is your protocol supported by Wireshark?
• What are the data types and representation format? Protocol states?
• Is authentication & encryption required?
• If authentication is required, can you replay?
Info Gathering

- Protocol specification (partial)
  - [http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol](http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol)
- Not supported by Wireshark
  - PDML doesn’t help me here
  - So I need to use GPF or Taof
- No authentication by default
- Mixed Binary + ASCII protocol
- Passes lots of JavaScript/JSON
  - Fusil might be a possibility here
- Build on existing client implementations?
20 Minute Results

• Taof
  – Used proxy mode to connect mongo client to server
  – Logged initial connection

• GPF
  – Server rejected all payloads generated by “simple fuzzing” - bad recv() mostly due to length
  – Converted login sequence and used replay mode
    • Many caught assertions in BSON processing and assertion failures
    • Created “interesting” databases and eventually a malloc failure
CONCLUSIONS
Non-Surprising Conclusions

• There is no single fuzzer (or framework) to “rule them all”
  – All of the tools have tradeoffs & feature/documentation gaps

• Seemingly dead projects (and even those written in C) can still be useful

• Pay me now or may be later
  – You will have to write “code” no matter what
  – Ambivalent about learning/using block-based fuzzing DSLs
  – Generation & mutation is not the only thing you do with the protocols
So going forward...

• For quick best-effort fuzzing, go with GPF
  – or Taof for fuzzing newbies

• Develop protocol specific fuzzers in Python but re-use APIs where possible
  – Sulley, Antiparser, and possibly even Peaches
A Subjective Fuzzer “Magic Quadrant”

- isic
- tcpject
- taof
- antiparser
- ruckus
- GPF
- fusil
- sulley
- Peach