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

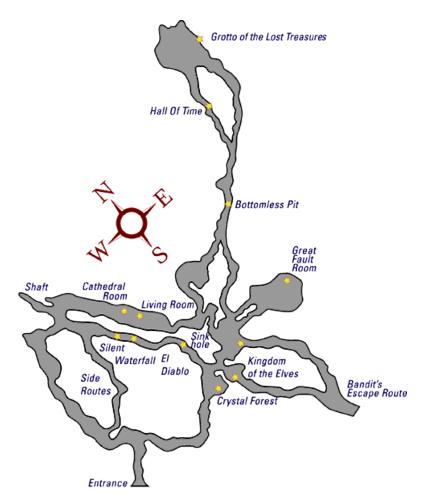
Matthew Franz

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)



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 blockbased 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 nontrivial % of vulns discovered
 - Even integrated into commercial singature 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 welldefined 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 a 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

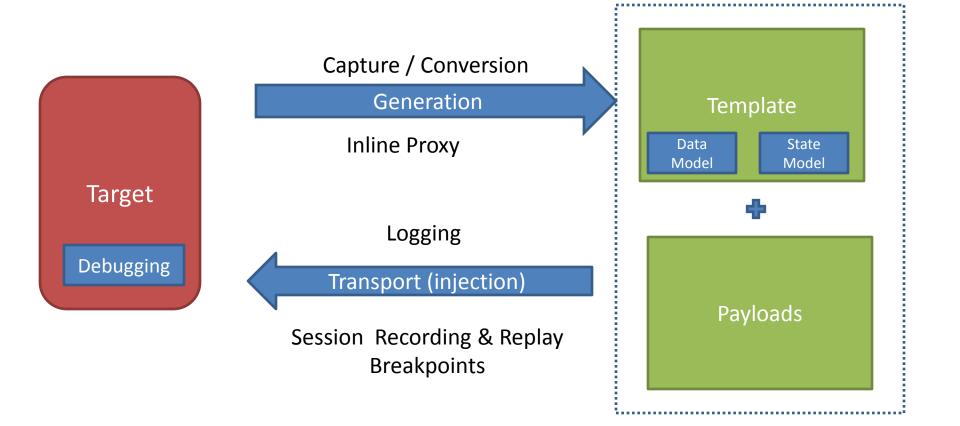
- Target external interface under test
 - Client, Server, Parser, Kernel, Protocol, etc.
- Mode of Operation
 - API
 - Executable
- Language Python, C, Ruby, etc.
- Transport you can inject test cases into the application/protocol (TCP, IP, UDP, SSL, IPv6)
- Template
 - Generation manual automated, inline, 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)



Mode of Operation

Operating Modes

- Approaches
 - API-based
 - Write code in a scripting language
 - Extend existing processers
 - 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

File Edit View Windows Help

```
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
      );
                 : modbus tcp.unit id
      // name
      // showname: unit identifier: 1
      // show
               : 1
      // size: 0x1 (1)
      hex(
      01
```

0000000053 6F 75 72 63 65 3A 43 20 53 69 7A 65 3A 30 30 37 30 Source:C Size:0070 0000001220 44 61 74 61 3A 46 00 00 00 C7 57 47 73 00 00 00 00 Data:F....WGs.... 00000024D4 07 00 00 00 00 00 00 61 64 6D 69 6E 2E 24 63 6D 64\$cmd 0000003600 00 00 00 00 FF FF FF FF 1F 00 00 00 03 71 75 65 72 0000004879 00 13 00 00 00 10 69 73 6D 61 73 74 65 72 00 01 00 v....ismaster... 0000005A00 00 00 00 53 6F 75 72 63 65 3A 53 20 53 69 7A 65 3ASource:S Size: 0000006C30 30 38 37 20 44 61 74 61 3A 57 00 00 00 69 26 83 C2 0087 Data:W...i&.. .WGs..... 0000009000 00 00 00 00 00 01 00 00 00 33 00 00 00 10 69 73 6Dism 000000A261 73 74 65 72 00 01 00 00 00 02 6D 73 67 00 0B 00 00 aster....msg.... 000000B400 6E 6F 74 20 70 61 69 72 65 64 00 01 6F 6B 00 00 00 .not paired..ok... 000000C600 00 00 00 F0 3F 00 ? . Signed 32 bit: 1920298835 Signed 8 bit: 83 Hexadecimal: 53 Unsigned 8 bit: 83 Unsigned 32 bit: 1920298835 123 Octal: Signed 16 bit: 28499 32 bit float: 4.861338e+30 Binary: 01010011 Unsigned 16 bit: 28499 64 bit float: 7.429827e+15 Stream Length: 8 ✓ Show little endian decoding Show unsigned and float as hexadecimal Offset: 0

Autodafe (Modbus/TCP)

GPF (MongoDB)

A Peach Template

<!-- Create a simple data template containing a single string -->
<DataModel name="HttpRequest">

<!-- The HTTP request line: GET http://foo.com HTTP/1.0 -->
<Block name="RequestLine">

<!-- Defaults can be optionally specified via the value attribute --> <String name="Method"/> <String value=" " type="char"/> <String name="RequestUri"/> <String value=" "/> <String name="HttpVersion"/> <String value="\r\n"/>

</Block>

<String value="\r\n"/>

<Blob name="Body" minOccurs="0" maxOccurs="1"/>

```
</DataModel>
```

<StateModel name="State2" initialState="Initial">
 <State name="Initial">
 <State name="Initial">
 <State name="Initial">
 </state="Output">
 </Data Model ref="HttpRequest" />
 <Data ref="HttpDptions" />
 </Action>
 </State>
</StateModel>

<!-- Create a simple test to run --> <Test name="HttpGetRequestTest" description="HTTP Request GET Test"> <StateModel ref="State1"/> <!-- Target a local web server on port 80 --> <Publisher class="tcp.Tcp"> <Param name="host" value="127.0.0.1" /> <Param name="port" value="80" /> </Publisher> </Test> <Test name="HttpOptionsRequestTest" description="HTTP Request OPTIONS Test"> <StateModel ref="State2"/> <!-- Target a local web server on port 80 --> <Publisher class="tcp.Tcp"> <Param name="host" value="127.0.0.1" /> <Param name="port" value="80" /> </Publisher> </Test> <!-- Configure a single run -->

```
<Run name="DefaultRun" description="HTTP Request Run">
```

<!-- The set of tests to run -->
<Test ref="HttpGetRequestTest" />
<Test ref="HttpOptionsRequestTest" />

</Run>

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
 - Рсар
 - 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)
- SPIKE (4/2004)
- 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

*	Tcpjunk (as superuser)
File Edit Tags Help	······································
Host/IP fe80::222:15ff:fe15:16d	4%eth1 Service/Port http://IPv4/IPv6//SSL///Connect 🔯 Accept 🛛 📰 Settings
Wait for data at first	untitled blabla
Don't wait for data at last	GET /blablaba HTTP/1.0
Connection timeout (s) 2	
Session timeout (ms) 100	
Repeat session 1	
Source port 666	
Source IP	GET /blablaba HTTP/1.0
2002:0000000	HTTP/1.1 404 Not Found
Source IP interface eth1	Date: Sat, 26 Dec 2009 21:58:18 GMT Server: Rpache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.81 DRV/2
send buffer size	Vary: accept-language,accept-charset Accept-Ranges: bytes
send delay (ms)	Connection: close
SSL server certificate	Content-Type: text/html; charset=iso-8859-1 Content-Language: en
(None)	Expires: Sat, 26 Dec 2009 21:58:18 GMT
SSL server key	<pre><?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"</pre> </pre>
(None)	"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml"></html>
Hex dump	<pre>(head) <title>Object not found!</title></pre>
	k rev="made" href="mailto:you@example.com" />
Auto scroll session log	<pre><style type="text/css"><!/*><![CDATA[/*><!*/ bodu (color: #000000; background-color: #FFFFF;)</pre></td></tr><tr><td>Auto clear status</td><td>Dody (Color: #000000; Dackground=Color: #FFFFFF;)</td></tr><tr><td>Highlight</td><td>Binding to IP 2002:129c:916d:57a1:91ee:96c:fbde:6e42</td></tr><tr><td>Session font</td><td>Connected to: fe80::222:15ff;fe15:16d4 Sent 24 bytes</td></tr><tr><td>Terminus 10</td><td>Received 1377 bytes</td></tr><tr><td>Terminus 10</td><td>Connection closed</td></tr></tbody></table></style></pre>

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

CAVEAT: <u>http://github.com/mongodb/mongo-c-driver/</u> does show evidence of embedded fuzzing in bson.c

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)
 - <u>http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol</u>
- 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?

000000000	4b	00	00	00	4b	1d	25	06	00	00	00	00	d4	07	00	00	Κ	.K.%.		
00000010	00	00	00	00	61	64	6d	69	6e	2e	24	63	6d	64	00	00		.admi	n.\$	cmd
00000020	00	00	00	ff	ff	ff	ff	24	00	00	00	03	71	75	65	72		\$.quer
00000030	79	00	18	00	00	00	10	6C	69	73	74	44	61	74	61	62	у	l	ist	Datab
00000040	61	73	65	73	00	01	00	00	00	00	00						ase	s		
00000	900	_f3	00	00	00	6a	26	83	c2	4b	10	25	06	01	00	00	00		j&	K.%
00000	910	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00 (
00000	920	01	00	00	00) cf	00	00	00	04	64	61	. 74	61	62	61	73			.databas
00000	930							00			00	30	00	00	00	02	2 6e	es		0.0n
00000	940							00			72	74	30	00	01	. 73	69	ame.		art0si
00000								73												
00000								79												1.1
00000	970							00												admin
00000								4f												Disk
00000		00						65												pty2
00000		00						6e	~											mel
00000		6f						73												zeOnDisk
00000								00												?.empty.
00000								74												lSize
00000		00				94	41	01	6f	6b	00	00	00	00	00	00	00		.A.o	k
00000	9F0	fe) 3f	00)													.?.		

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"

