Harder, Better, Faster, Stronger

Semi-Auto Vulnerability Research
Professional Vulnerability Research

- Finding bugs is not the problem
  - Fuzzing works
    - Microsoft found over 1800 bugs in Office 2010
    - 280 bugs found in Mozilla JavaScript using JSFunFuzz
      - [https://bugzilla.mozilla.org/show_bug.cgi?id=jsfunfuzz](https://bugzilla.mozilla.org/show_bug.cgi?id=jsfunfuzz)

- Tooling is not the problem
  - Distributed fuzzing
  - Crash analyzers

- Lack of intelligent workflow is the problem
Main Goal

Develop an effective workflow and toolset for fuzzing and triaging vulnerabilities in a production environment
Ancillary Goals

● Primary
  ▶ Determine cause and exploitability
  ▶ Human time efficiency

● Secondary
  ▶ CPU efficiency
  ▶ Ease of use
Process Breakdown

- Attack Surface Analysis
- Input Selection
- Fuzzing
- Triage
Keys to Fuzzing Smartly

● Input selection
  ▶ Most important factor in timely bug discovery
  ▶ Time management

● Automation
  ▶ SIMPLE Distributed fuzzing
  ▶ Crash analysis
  ▶ Bucketing
  ▶ Confidence Rating
Keys to Smart Bug Triage

● Crash selection
  ▶ Select for understanding
  ▶ Crash database
  ▶ Bug classes

● Program flow analysis
  ▶ Code coverage
  ▶ Input Mapping
  ▶ Taint Analysis
Input Selection

● Attack Surface Analysis
  ▶ Determine which areas of the code are reachable from external inputs

● Template code coverage
  ▶ Determine what areas of code are exercised by different templates

● Rank templates based upon coverage of targeted code or overall attack surface
Fuzzing

The Miller Theorem

\[ C = \text{code path coverage} \]
\[ T = \text{Time spent Fuzzing} \]
\[ B = \text{Bugs Discovered} \]

\[ \partial C \cdot \partial T = \partial B \]
Fuzzing

- Obey the Miller Theorem
  - Create inputs to maximize coverage
  - Create the framework to maximize uptime

- Generation vs. Mutation
  - If you can, do both!
  - Mutation is cheaper, still works

- Do as little work as possible
  - Re-do as little work as possible
Fuzzing

- Test Execution
- Data Collection
- Retesting
Fuzzing – Test Execution

● Watch your tests well
  ▶ Embedded custom debugger
  ▶ Be able to gather needed data at crash time
  ▶ Make use of debugging technologies
  ▶ Be able to avoid invalid exceptions

● Distribute your tests
  ▶ Centralized management
  ▶ Make it easy to add nodes
Fuzzing – Data Storage

- Use a database!
  - Store lots of data over time
  - Easily searched

- What to store
  - Store what you need for crash selection
  - All crash information
  - Software versioning information
    - Binary diffs
Fuzzing - Retesting

● Maintaining a good database allows:
  ▶ Automated retesting of modified code paths
  ▶ Automated retesting of crashes in modified code paths

● Track bug life across software versions
  ▶ A bug which lives through a nearby patch can have a long shelf-life
    ● MS08-067 and MS06-040
    ● ANI
Triage – Crash Selection

- Which crashes should receive priority?

- What properties make crashes more exploitable?
  - Knowledge! Familiarity!

- Crash database
  - Vulnerability properties
  - Searchable crash criteria
Triage – Crash Selection

- **Exception Analysis**
  - Determine level of access exception grants user

- **Bug Class Identification**
  - Difficulty of exploitability varies by bug class
  - Custom architecture problems
    - Custom memory allocators
Triage – Program Flow Analysis

● Abstract a program into flows
  ▶ Code execution
  ▶ Data dependency

● Code Coverage
  ▶ Block hit trace for path to exception
  ▶ Build a graph of program execution
  ▶ Augment static program graphs
Triage – Program Flow Analysis

- **Input Mapping**
  - Trace APIs or System Calls that perform I/O
  - Mark data copied from external sources into memory

- **Taint analysis**
  - Follow input through the execution of the program
  - Determine where the bytes of the crash originated
  - Potential for exploit and signature generation
Triage – Program Flow Analysis

- Visualization
  - Provides a graphical representation of program structure and execution paths
  - Visualization allows overlaying multiple graphs and datasets using visual cues
  - Converting data to a visual problem allows rapid understanding of large datasets
Moflow: Input Selection
Input Selection - Requirements

- Attack Surface Analysis
  - Call graph analysis

- Template code coverage
  - Dynamic tracing

- Template ranking
  - Coverage graph analysis
## Attack Surface Analysis

- Obtain call graph
  - IDA2Moflow.idc
  - LibCodis

- Define APIs that are data entry points

<table>
<thead>
<tr>
<th>Input Source</th>
<th>I/O API</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td></td>
</tr>
<tr>
<td>NtOpenFile()</td>
<td>NtReadFile()</td>
</tr>
<tr>
<td>NtCreateFile()</td>
<td>NtWriteFile()</td>
</tr>
<tr>
<td>SYS_Open()</td>
<td>SYS_Read()</td>
</tr>
<tr>
<td></td>
<td>SYS_Write()</td>
</tr>
<tr>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>connect()</td>
<td>send()</td>
</tr>
<tr>
<td>accept()</td>
<td>recv()</td>
</tr>
</tbody>
</table>
Attack Surface Analysis

- Determine reachability graph from each API

\[
\begin{align*}
\delta\text{-wavefront} & \leftarrow \text{RootSet} \\
closure & \leftarrow \emptyset \\
\text{While } \text{nonEmpty}(\delta\text{-wavefront}) \text{ Do} \\
\quad \text{wavefront} & \leftarrow \text{oneStep}(\delta\text{-wavefront}) \\
\quad \delta\text{-wavefront} & \leftarrow \text{wavefront} - \text{closure} \\
\quad \text{closure} & \leftarrow \text{closure} \cup \delta\text{-wavefront} \\
\text{End While} \\
\text{Return closure}
\end{align*}
\]

\(\delta\text{-wavefront} \text{ Algorithm – Qadah et al.}\)
Template Code Coverage

- **Dynamic Tracing**
  - Instrument each basic block in a program
  - Efficiently record execution order of all blocks

- **Implementation - PinFlow**
  - Program tracer written as a PinTool
  - Hook on block cache creation
  - Inject instructions into cached code blocks
  - Callback function writes binary struct to ringbuffer
  - Ringbuffer flushed when full and on program exit
Template Code Coverage

- Moflow Visualizer PinFlow Trace Launcher
Template Code Coverage

- Advantage – Speed
  - PIN is much faster than traditional breakpoint or trap based solutions

<table>
<thead>
<tr>
<th>7zip Benchmark Test</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Tracer</td>
<td></td>
</tr>
<tr>
<td>Process Stalker</td>
<td>20.48</td>
</tr>
<tr>
<td>PinFlow</td>
<td>1.77</td>
</tr>
</tbody>
</table>

11.57 times faster!
Template Prioritization

- Select functions for attack surface

- Calculate reachability to create attack surface graph

- Rank stored traces by number of nodes hit in attack surface graph
Template Prioritization

```
Mflow Visualizer v0.2

Project Tree
- Modules
  - acord32.dll.xml
- Block Traces
  - Traces\acord32.dll-pdf-0001-flowTrace.xml
  - Traces\acord32.dll-pdf-0002-flowTrace.xml
  - Traces\acord32.dll-pdf-0003-flowTrace.xml
  - Traces\acord32.dll-pdf-0004-flowTrace.xml
  - Traces\acord32.dll-pdf-0005-flowTrace.xml
  - Traces\acord32.dll-pdf-0006-flowTrace.xml
  - Traces\acord32.dll-pdf-0007-flowTrace.xml
  - Traces\acord32.dll-pdf-0008-flowTrace.xml
  - Traces\acord32.dll-pdf-0009-flowTrace.xml
  - Traces\acord32.dll-pdf-0010-flowTrace.xml
  - Traces\acord32.dll-pdf-0011-flowTrace.xml
  - Traces\acord32.dll-pdf-0012-flowTrace.xml
- Input Files
  - Inputs\pdf0001.xml
  - Inputs\pdf0002.xml
  - Inputs\pdf0003.xml
  - Inputs\pdf0004.xml
  - Inputs\pdf0005.xml
  - Inputs\pdf0006.xml
  - Inputs\pdf0007.xml
  - Inputs\pdf0008.xml
  - Inputs\pdf0009.xml
  - Inputs\pdf0010.xml
  - Inputs\pdf0011.xml
  - Inputs\pdf0012.xml

Status Window | Trace Explorer | Trace Ranking

Attack Surface Graph
- acord32.dll-pdf-attack-surface

Traces

<table>
<thead>
<tr>
<th>Name</th>
<th>Blocks</th>
<th>Edges</th>
<th>Targeted %</th>
</tr>
</thead>
<tbody>
<tr>
<td>acord32.dll-pdf-0001</td>
<td>3141</td>
<td>3985</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0002</td>
<td>3036</td>
<td>3284</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0003</td>
<td>3141</td>
<td>3985</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0004</td>
<td>3570</td>
<td>4221</td>
<td>5</td>
</tr>
<tr>
<td>acord32.dll-pdf-0005</td>
<td>3141</td>
<td>3985</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0006</td>
<td>3141</td>
<td>3985</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0007</td>
<td>3915</td>
<td>4414</td>
<td>12</td>
</tr>
<tr>
<td>acord32.dll-pdf-0008</td>
<td>3798</td>
<td>4023</td>
<td>66</td>
</tr>
<tr>
<td>acord32.dll-pdf-0009</td>
<td>1867</td>
<td>2017</td>
<td>0</td>
</tr>
<tr>
<td>acord32.dll-pdf-0010</td>
<td>3772</td>
<td>3473</td>
<td>100</td>
</tr>
<tr>
<td>acord32.dll-pdf-0011</td>
<td>3772</td>
<td>3473</td>
<td>100</td>
</tr>
<tr>
<td>acord32.dll-pdf-0012</td>
<td>3772</td>
<td>3473</td>
<td>100</td>
</tr>
</tbody>
</table>
```
Graph Visualization

Moflow Block Trace Graph Visualization
Fuzzing Automation
Fuzzing Automation

- Distributed Fuzzing
- Fuzzer Management
- Data Gathering
- Crash Mining
Distributed Fuzzing

- Tests are small and atomic
  - Distribute simply
  - Make it easy to add systems
  - Easy to add tests

- Centralized Management
  - Aids in speedy addition of hardware
Fuzzer Management

● Customizable yet simple
  ▶ Ignore first chance exceptions?
  ▶ Add debugging technologies?
  ▶ Max test case timeout

● Ease of use is key
  ▶ Quick recovery for dead hosts
  ▶ Quick addition of new hosts
  ▶ Centralized management w/ database
Fuzzer Management

- Jobs are held in the central DB
  - Job details passed to workers
  - Test cases are generated by workers as needed
  - Successful crashes are returned to the DB with details

- Test cases are wrapped with a custom debugger

- Data is returned to the central DB
Basic Worker

- Job Collection (curl -> sql database)
- Fuzzing Engine (anything you like!)
  - CPU Monitor (WBEM)
  - Debugged Test Thread (dbgext)
  - Data Collection (Codis, dbgext)
  - Data Reporting (curl -> sql database)
Data Gathering

● Store what you must
  ▶ Bucketing
  ▶ Categorization
  ▶ Indicators of Exploitability

● Store what you have
  ▶ Why redo work?
  ▶ Can’t know what you may need

● Store it smart
  ▶ Database!
Crash Mining

- Post-crash analysis is performed on crashes deemed “relevant”

- Relevant crashes are those which are:
  - Familiar to your exploit developers
  - Relate to your attacking goals

- Relevant crashes are mined as needed from the database with queries.
  - What is relevant changes over time.
## Jobs

<table>
<thead>
<tr>
<th>Name</th>
<th>Extension</th>
<th>State</th>
<th>Buckets</th>
<th>Offset</th>
<th>Try</th>
<th>Edit</th>
<th>Delete</th>
<th>Show</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>mp3</td>
<td>Paused</td>
<td>1</td>
<td>15,505</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>5</td>
<td>11,594</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>avi</td>
<td>Completed</td>
<td>1</td>
<td>1,527,167</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Buckets for c:\Program Files\Windows Media Player\wmplayer.exe

<table>
<thead>
<tr>
<th>Name</th>
<th>Crashes</th>
<th>Notes</th>
<th>Bucket sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000000000000</td>
<td>1564</td>
<td>0</td>
<td>Sample</td>
</tr>
</tbody>
</table>

**1 Found**

<table>
<thead>
<tr>
<th>Name</th>
<th>Extension</th>
<th>State</th>
<th>Bucket sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
<tr>
<td>c:\Program Files\Windows Media Player\wmplayer.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
<tr>
<td>c:\Users\Wrt\Desktop\brutefile\testpattern.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
<tr>
<td>c:\Users\Wrt\Desktop\brutefile\testpattern.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
<tr>
<td>c:\Users\Wrt\Desktop\brutefile\testpattern.exe</td>
<td>mp4</td>
<td>Paused</td>
<td>Sample</td>
</tr>
</tbody>
</table>

### Buckets for c:\Users\\Wrt\\Desktop\\brutefile\\testpattern.exe

<table>
<thead>
<tr>
<th>Name</th>
<th>Crashes</th>
<th>Notes</th>
<th>Bucket sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffffdf0000000000000000000000000000000000</td>
<td>47</td>
<td>0</td>
<td>Sample</td>
</tr>
</tbody>
</table>

**Crashes for ffffdf0000000000000000000000000000000000**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Offset</th>
<th>Try</th>
<th>Sample</th>
<th>Show</th>
</tr>
</thead>
<tbody>
<tr>
<td>00401058 cc int 3</td>
<td>100</td>
<td>ADD 4 64</td>
<td>-</td>
<td>Show</td>
</tr>
</tbody>
</table>

### Show Crash

**Instruction**

```
00401058 cc int 3
```

**Information**

```
eax=00000000 ebx=7fffd900 ecx=004011d5 edx=7777064f4 esi=00000000 edi=00000000
eip=00401058 esp=00127f14 ebp=00127f40 iopl=0
cs=*011b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000026

00401058: int3
```

**Offset**

```
100
```

**Try**

```
REPLACE 1 H 7F
```

**Sample**

-
Moflow: Triage
Triage - Requirements

- Exploitability
  - Exception information
  - Deep Trace

- Triggering Condition
  - Fuzzer feedback
  - Taint analysis

- Root Cause
  - Graph analysis
Triage - Exploitability

● Exception Information
  ▶ Brutefile outputs XML data containing exception information

● Deep Trace
  ▶ Code Coverage
  ▶ Attack surface APIs
  ▶ Dataflow
Triage - Exploitability

● Dataflow
  ▶ Once exception is found program is traced using PinFlow to gather instruction level instrumentation
  ▶ Blocks are hooked during cache and disassembled to instrument instructions that access memory
  ▶ Dataflow callback function records the address and value of each memory read or write

● Taint Analysis
  ▶ Provides exception analysis functions with information about controlled bytes
  ▶ Knowledge of controlled bytes allows more precise analysis
Triage – Triggering Condition

- **Fuzzer Feedback**
  - As part of exception analysis data Brutefile includes information about mutation

- **Taint analysis**
  - When triaging a bug from input with unknown modifications, perform taint analysis
  - Forward taint propagation from memory allocated to stored data from input file will reveal which bytes are referenced in the exception
Triage – Root Cause

● Graph Analysis
  ▶ Overlay graphs of several deep traces to determine similarity
  ▶ If execution trace leading up to the crash is identical but different bytes were manipulated, root cause should be determined

● Taint analysis
  ▶ Follow tainted data in the exception back to the code location that first influenced the memory location with external data
Moflow: Tools
Console Disassembler

- Console interface for libcodis

Static Analysis

- Instruction Disassembly
- Function Detection
- Code and Data Cross-References
- Function Control Flow Graph
- Call Graph

- Import IDA2Moflow and .map files
Windbg Integration

- CodisExt
  - Windbg extension using the engextcpp API
  - Utilizes libcodis to extract disassembly graphs and cross-references
  - Utilizes Windbg DML functionality to allow a hyperlinked interface for cross references
Windbg Integration

0:000> !codis
[codis] Usage:
[codis] !codis load <moduleName> Load a module into the disassembler engine
[codis] !codis xrefs [functionAddr] Show caller/callees
[codis] !codis callers <functionAddr> Show function callers
[codis] !codis callees <functionAddr> Show function callees
[codis] !codis names Show names in codis database
[codis] !codis dis <moduleName> [functionAddr] Dump disassembly of a module or function
[codis] !codis dot Dump a GraphViz DOT file

0:000> !codis load test
[codis] Loading C:\Vulndev\test.exe

;------------------------------------------------------------------
; File Header
;------------------------------------------------------------------
; Binary format: 32-bit PE
; Byte Ordering: Little Endian
; Entry Point: 0000130b
; File Size: 112128 bytes
;------------------------------------------------------------------
Windbg Integration

0:000> !codis xrefs
[codis] Function: 00401005 sub_00401005
[codis] xrefs to: 00401149
[codis] xrefs from:
[codis] Function: 0040100a sub_0040100a
[codis] xrefs to: 0040100f
[codis] xrefs from:

--- SNIP ---

[codis] Function: 00411850 sub_00411850
[codis] xrefs to: 00411763
[codis] xrefs from:
[codis] Function: 00411a58 wrapper_RtlUnwind
[codis] xrefs to: 0040e530 00407732
[codis] xrefs from:
[codis] Function: 44cbe836 sub_44cbe836
[codis] xrefs to: 0040e53

0:000> !codis dot
digraph G {
"00401005"
"0040100a"
"0040100f"
"004010c0"
"0040113a"

--- SNIP ---

"00401076" -> "0040100a"
"00401058" -> "0040113a"
"0040104b" -> "004010c0"
"0040100f" -> "00401070"
"0040100a" -> "00401030"
}
Windbg Integration

0:000> !codis dis test 00402eea
00402eea |
........ | ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ | ;; SUBROUTINE ;;;
........ | ;;;;;;;;;;;;;;;;;;
........ | sub_00402eea: ; xrefs: 0x00402f68 0x00402f31 0x004015f7
........ | 6a 08 | push byte 0x8
00402eec | 68 60 a2 41 00 | push dword 0x41a260
00402ef1 | e8 32 fd ff ff | call <sub_00402c28>
00402ef6 | e8 2e f9 ff ff | call <sub_00402829>
00402efb | 8b 40 78 | mov [eax+0x78], eax
00402efe | 85 c0 | test eax, eax
00402f00 | 74 16 | jz 0x402f18
00402f02 | 83 65 fc 00 | and dword 0x0, [ebp-0x4]
00402f06 | ff d0 | call eax
00402f08 | eb 07 | jmp 0x402f11
00402f0a | 33 c0 | xor eax, eax
00402f0c | 40 | inc eax
00402f0d | c3 | ret
00402f0e | 8b 65 e8 | mov [ebp-0x18], esp
00402f11 |
........ | loc_00402f11: ; xrefs: 0x00402f08
........ | c7 45 fc fe ff ff ff | mov dword 0xffffffff, [ebp-0x4]
00402f18 |
........ | loc_00402f18: ; xrefs: 0x00402f00
........ | e8 4e 48 00 00 | call <sub_00407763>
IDA Integration

● IDA2Moflow.idc
  ▶ Dumps static program call graph
    ● Module
    ● Functions
    ● Calls
  ▶ Works on all versions of IDA

● Useful to overcome current limitations in static analysis provided by libcodis
Questions?
● Email:  rjohnson@sourcefire.com
         richinseattle@gmail.com
● Twitter: Richinseattle
● Email:  lgrenier@sourcefire.com
         pusscat@metasploit.com
● Twitter: Pusscat
● Special Thanks to Chris McBee!
Extra Slides
Template Code Coverage

- Dynamic Tracing

- Implementation
  - Program tracer written as a PinTool
  - Designed for Win32 platform
  - Function and Block hooking for Code Coverage
  - System call hooking for I/O*
  - Memory reference trace*
  - Logging to standardized format
Static Analysis

- Instruction Disassembly
- Function Detection
- Code and Data Cross-References
- Function Control Flow Graph
- Module / Program Call Graph
Instruction Decoding

- BeaEngine 4
  - Multi-Architecture
    - x86 / x64
  - High performance
    - [stats]
  - Actively developed
    - [stats]
Function Detection

- Prologue Detection
  - [Image of prologues]

- Static call targets
  - [show dynamic call vs static call]
Code and Data Cross-References

- Disassembly of functions results in extraction of CALLs, JMPs, and static data references

- [image goes here]
Function Control Flow Graph

- Break a function into basic blocks
  - JMP
  - CALL
  - RET
Module / Program Graph

- Enumerate function cross references

- Support loading multiple modules for inter-modular call graph
Dynamic Analysis

- LibMoflow
  - High level program analysis library in C#
  - Code Coverage Analysis
  - Trace Differencing
  - Graph Analysis
  - Tainted Data Analysis
Code Coverage Analysis

- Augment graph from static analysis with code coverage
- Trace Differencing
- CrashViz
  - Program Graph
  - Trace Overlays
Trace Differencing

- Describe algorithm here
Graph Analysis

- Loop Detection
  - Dominator Trees
  - etc
File Visualization

- Hex and Structured Tree Views
- Visualize Fuzzer File Mutations and other session metadata
- Structure Decoding
  - Office Formats (GUT)
  - PDF (Only’s lib?)
  - FLASH (Patrick/Shong’s lib)