Fuzzing for Security Flaws

John Heasman

Stanford University, April 2009

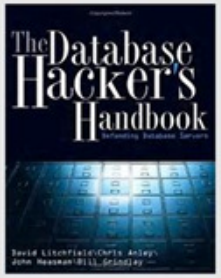
Agenda

- Introductions
- What is fuzzing?
- What data can be fuzzed?
- What does fuzzed data look like?
- When (not) to fuzz?
- Two approaches and a basic methodology
- Advanced techniques
- Unsolved challenges

Introduction: NGS

- Established 2001, acquired 2008 by NCC Group
- Core offerings:
  - Consultancy
  - Training
  - Research
  - Software Products
- World leaders in application assessment
  - Product assessment for vendor
  - Web application assessment for corporations

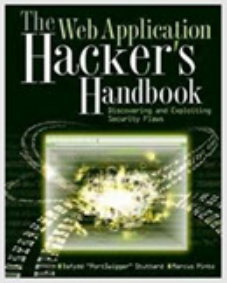
Publications



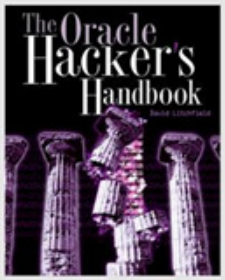
David Litchfield, Chris Anley, John Heasman, Bill Grindlay

David Litchfield, Chris Anley

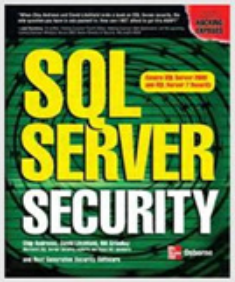
Chris Anley, John Heasman



Dafydd Stuttard, Marcus Pinto

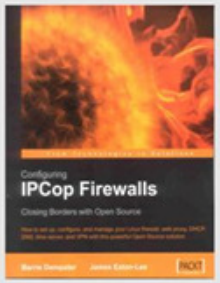


David Litchfield

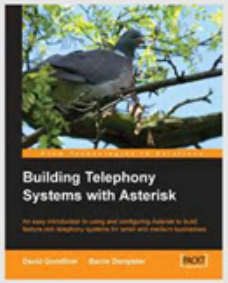


David Litchfield, Bill Grindlay

Barrie Dempster



Barrie Dempster



Barrie Dempster

About Me

- Masters in Engineering & CS from Oxford University
- Joined NGS London office in 2003
- Now VP of Research based in Seattle, WA
- Regular presenter at the Black Hat conferences
- Research interests:
  - Rootkits (ACPI, BIOS, PCI option ROMS, (U)EFI)
  - Web 2.0, Code execution vulnerabilities in Java
  - Vulnerability research

What is fuzzing?

- Feed target automatically generated malformed data designed to trigger implementation flaws
- A fuzzer is the programmatic construct to do this
- A fuzzing framework typically includes library code to:
  - Generate fuzzed data
  - Deliver test cases
  - Monitor the target
- Publicly available fuzzing frameworks:
  - Spike, Peach Fuzz, Sulley, Schemer
- Requirement of Microsoft's Secure Development Lifecycle program
- Still a long way to go - many vendors do no fuzzing!

What data can be fuzzed?

- Virtually anything!
- Basic types: bit, byte, word, dword, qword
- Common language specific types: strings, structs, arrays
- High level data representations: text, xml

Where can data be fuzzed?

- Across any security boundary, e.g.:
  - An RPC interface on a remote/local machine
  - HTTP responses & HTML content served to a browser
  - Any file format, e.g. Office document
  - Data in a shared section
  - Parameters to a system call between user and kernel mode
  - HTTP requests sent to a web server
  - File system metadata
  - ActiveX methods
  - Arguments to SUID binaries

What does fuzzed data consist of?

- Fuzzing at the type level:
  - Long strings, strings containing special characters, format strings
  - Boundary case byte, word, dword, qword values
  - Random fuzzing of data buffers
- Fuzzing at the sequence level
  - Fuzzing types within sequences
  - Nesting sequences a large number of times
  - Adding and removing sequences
  - Random combinations
- Always record the random seed!!

When to fuzz?

- Fuzzing typically finds implementation flaws, e.g.:
  - Memory corruption in native code
    - Stack and heap buffer overflows
    - Un-validated pointer arithmetic (attacker controlled offset)
    - Integer overflows
    - Resource exhaustion (disk, CPU, memory)
  - Unhandled exceptions in managed code
    - Format exceptions (e.g. parsing unexpected types)
    - Memory exceptions
    - Null reference exceptions
  - Injection in web applications
    - SQL injection against backend database
    - LDAP injection
    - HTML injection (Cross-site scripting)
    - Code injection

When not to fuzz

- Fuzzing typically does not find logic flaws
  - Malformed data likely to lead to crashes, not logic flaws
  - e.g. Missing authentication / authorization checks
- Fuzzing does not find design/repurposing flaws
  - e.g. A sitelocked ActiveX control with a method named “RunCmd”.
- However transitions in a state machine can be fuzzed...
  - Send well-formed requests out of order
  - But how to know when you’ve found a bug?

Two approaches

"Dumb"

- Fuzzer lacks contextual informational about data it is manipulating
- May produce totally invalid test cases
- Up and running fast
- Find simple issues in poor quality code bases

"Smart"

- Fuzzer is context-aware
  - Can handle relations between entities, e.g. block header lengths, CRCs
- Produces partially well-formed test cases
- Time consuming to create
  - What if protocol is proprietary?
- Can find complex issues

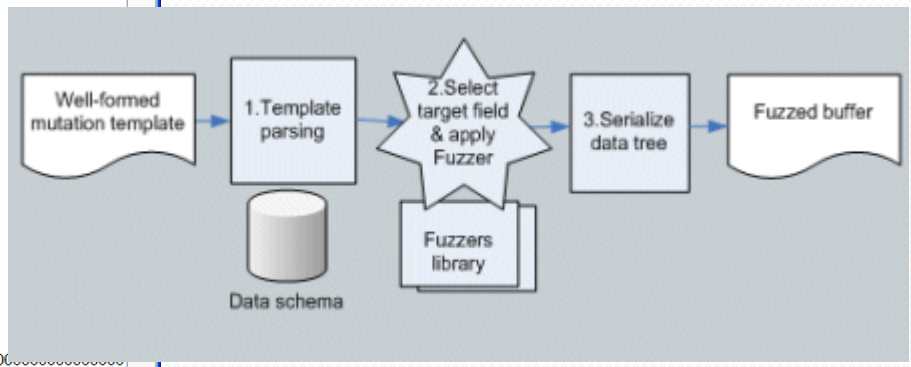
Pseudo-code for dumb fuzzer

```
for each {byte|word|dword|qword} aligned location in file
  for each bad_value in bad_valueset
  {
       file[location] := bad_value
       deliver_test_case()
  }
```

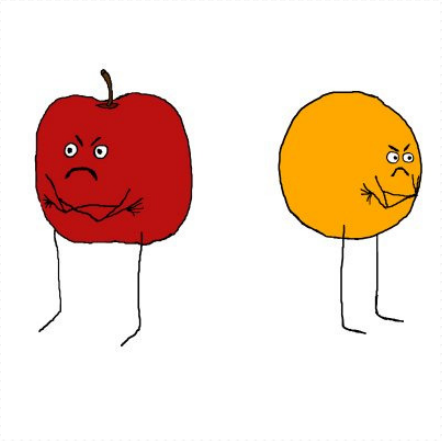
Sample config for smart fuzzer (1)

```
...
    o_jpeg =        fz3AddObjectToList( NULL, TYPE_BYTE, PTR(0xff), 1 ); // new header
                    fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xd8), 1 ); // unknown type
(start of file?)
                    fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xff), 1 );  // new header
                    fz3AddObjectToList( o_jpeg, TYPE_BYTE, PTR(0xe0), 1 );  // extension app0
marker segment
    o_jfif_len =    fz3AddObjectToList( o_jpeg, TYPE_WORD, BE_W(0x10), 2 ); // length
    o_jfif =        fz3AddObjectToList( o_jpeg, TYPE_COLLECTION, NULL, 0 );
    o_jfif_dat =    fz3AddObjectToList( NULL, TYPE_COLLECTION, NULL, 0 );
                    fz3AddObjectToList( o_jfif_dat, TYPE_STATIC, "JFIF", 5 );   // APP0 marker
                    fz3AddObjectToList( o_jfif_dat, TYPE_WORD, BE_W(0x0102), 2 ); // version
                    fz3AddObjectToList( o_jfif_dat, TYPE_BYTE, PTR(0), 1 );  // units
                    fz3AddObjectToList( o_jfif_dat, TYPE_WORD, BE_W(0x64), 2 ); // x density
                    fz3AddObjectToList( o_jfif_dat, TYPE_WORD, BE_W(0x64), 2 ); // y density
                    fz3AddObjectToList( o_jfif_dat, TYPE_BYTE, PTR(0), 1 );  // x thumbnail
                    fz3AddObjectToList( o_jfif_dat, TYPE_BYTE, PTR(0), 1 );  // y thumbnail
…
fz3AddAdditionalDataToObject( o_jfif, TYPE_COLLECTION, (BYTE *)o_jfif_dat, sizeof(object *) );
            …
            fz3SetObjectCallback( o_jfif_len, JPEG_set_length, o_jfif );
            …
```

Sample config for smart fuzzer (2)

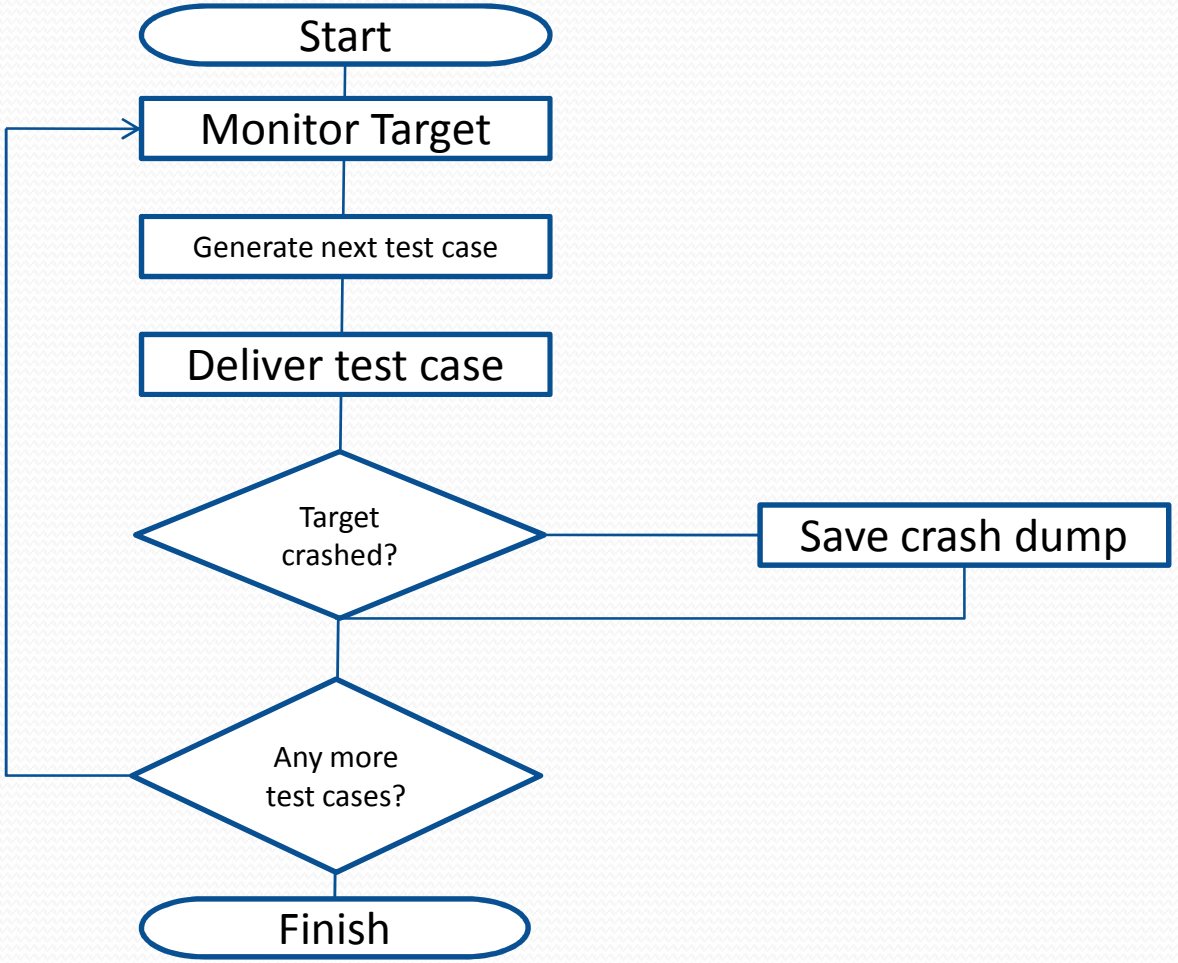


Two approaches cont.



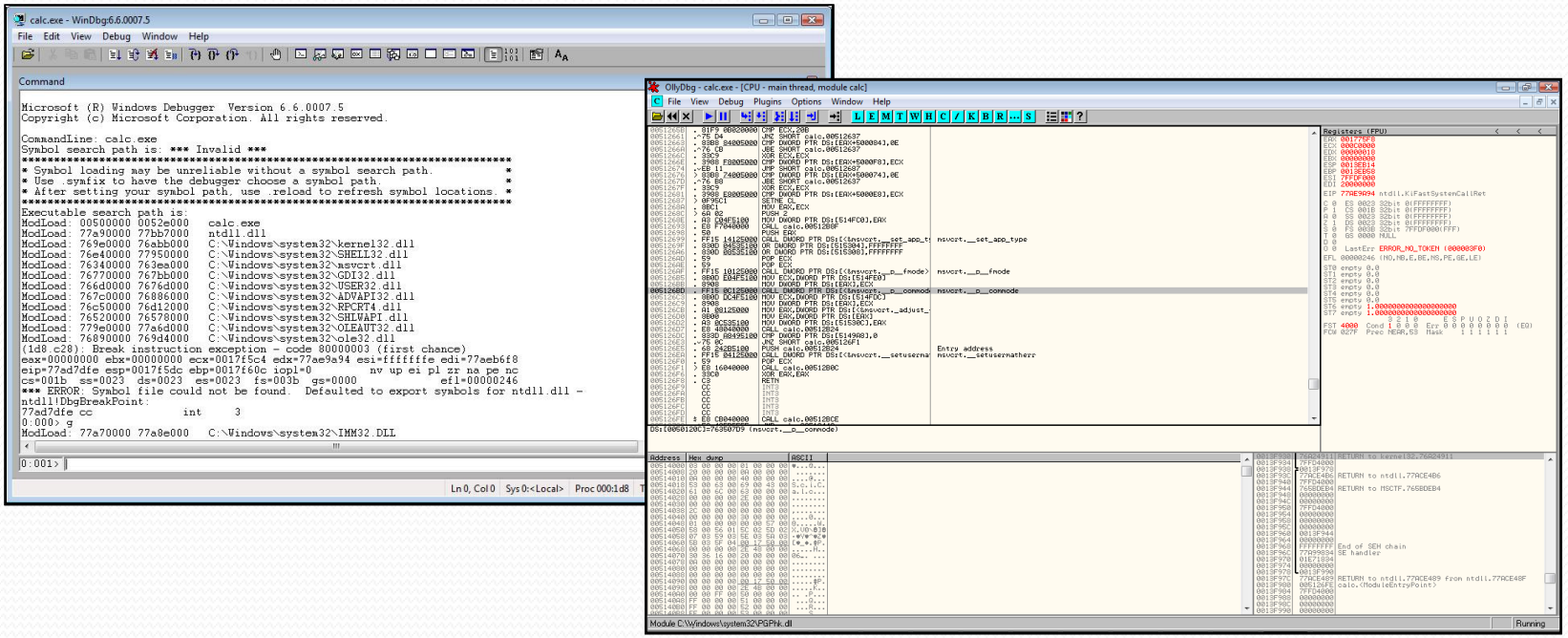
- Which approach is better?
- Depends on:
  - Time: how long to develop and run fuzzer
  - [Security] Code quality of target
  - Amount of validation performed by target
    - Can patch out CRC check to allow dumb fuzzing
  - Complexity of relations between entities in data format
- Don’t rule out either!
  - My personal approach: get a dumb fuzzer working first
  - Run it while you work on a smart fuzzer

Fuzzing in practice: the basic steps



Monitoring the target

1. Attach a debugger
   - Leverage existing functionality
   - Scripting, logging, crash dumps etc.



Monitoring the target

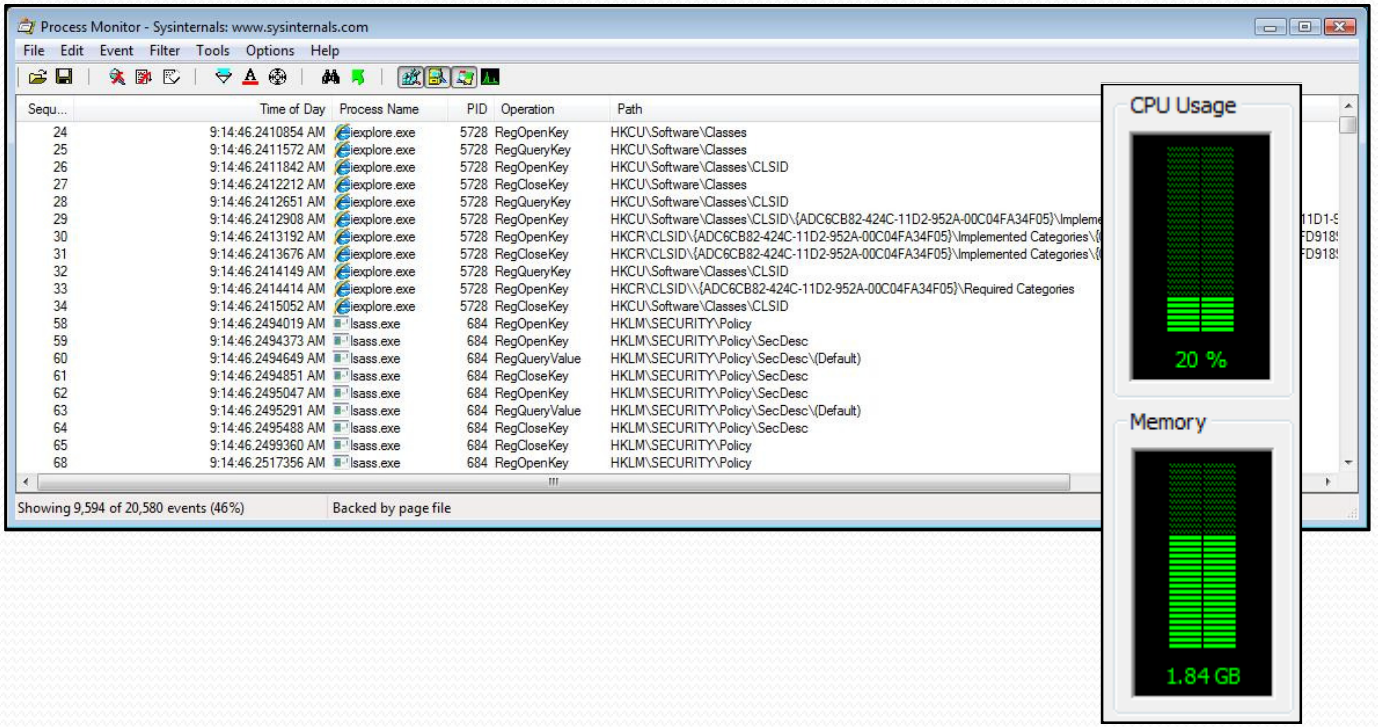
2. Write your own debugger
   - Actually easy to do
   - Lightweight, fast, full control

```
C++
BOOL WINAPI WaitForDebugEve
  __out   LPDEBUG_EVENT lpD
  __in    DWORD dwMilliseco
);
```

```
typedef struct _DEBUG_EVENT { /* de */
DWORD dwDebugEventCode;
DWORD dwProcessId;
DWORD dwThreadId;
union { EXCEPTION_DEBUG_INFO Exception;
CREATE_THREAD_DEBUG_INFO CreateThread;
CREATE_PROCESS_DEBUG_INFO CreateProcess;
EXIT_THREAD_DEBUG_INFO ExitThread;
EXIT_PROCESS_DEBUG_INFO ExitProcess;
LOAD_DLL_DEBUG_INFO LoadDll;
UNLOAD_DLL_DEBUG_INFO UnloadDll;
OUTPUT_DEBUG_STRING_INFO DebugString; }
u; } DEBUG_EVENT, *LPDEBUG_EVENT;
```

Monitoring the target

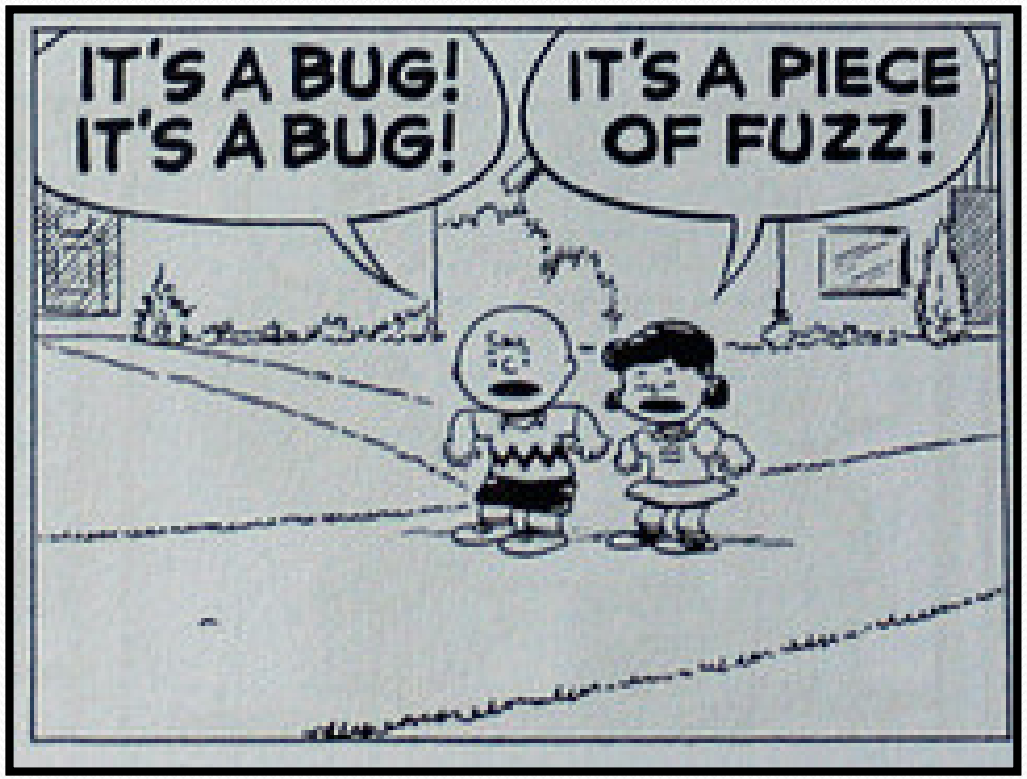
3. Monitor resources:
   - File, registry, memory, CPU, logs



Deliver the test case

1. Standalone test harness
   - E.g. to launch to client application and have it load fuzzed file format
2. Instrumented client
   - Inject function hooking code into target client
   - Intercept data and substitute with fuzzed data
   - Useful if:
     - State machine is complex
     - Data is encoded in a non-standard format
     - Data is signed or encrypted

Determining exploitability



Determining exploitability

- This process requires experience of debugging security issues, but some steps can be taken to gain a good idea of how exploitable an issue is…
- Look for any cases where data is written to a controllable address – this is key to controlling code execution and the majority of such conditions will be exploitable
- Verify whether any registers have been overwritten, if they do not contain part data sent from the fuzzer, step back in the disassembly to try and find where the data came from.
- If the register data is controllable, point the register which caused the crash to a page of memory which is empty, fill that page with data (e.g., ‘aaaaa…’)
- Repeat and step through each operation, until another crash occurs, reviewing all branch conditions which are controlled by data at the location of the (modified) register to ensure that they are executed

Determining exploitability

- Are saved return address/stack variables overwritten?
- Is the crash in a heap management function?
- Are the processor registers derived from data sent by the fuzzer (e.g. 0x61616161)?
- Is the crash triggered by a read operation?
- Can we craft a test case to avoid this?
- Is the crash triggered by a write operation?
- Do we have full or partial control of the faulting address?
- Do we have full or partial control of the written value?

Determining exploitability

- Recent advances – Microsoft !Exploitable
  - WinDbg extension to determine exploitability
  - Does lightweight taint analysis
  - Uses meta-instructions (STACK_PUSH, DATA_MOVE etc.)
  - Demoed at CanSec West conference, March 2009
  - http://www.microsoft.com/security/msec/default.mspx

A quick diversion

- These days you can get paid for finding vulnerabilities:
  - iDefense, Tipping Point, Pwn2Own
- Write a fuzzer, find bugs, $$$!!
- Here's how to get started:
  - Download FileFuzz (iDefense)
  - Pick a media application and find some samples
  - Fuzz! You will find bugs…

NGS fuzzed RealPlayer

Heap overflow triggered via malformed ID3v2 Tag in MP3 (reported by John Heasman) http://service.real.com/realplayer/security/10252007_player/en/
Buffer overflow triggered via malformed Mimio boardcast file (reported by John Heasman) http://service.real.com/realplayer/security/03162006_player/en/
Buffer overflow triggered via malformed RJS skin (reported by John Heasman) http://service.real.com/help/faq/security/security111005.html
HTML content within MP3 allows execution from local zone (reported by John Heasman) http://www.service.real.com/help/faq/security/security062305.html
Buffer overflow within DUNZIP32.dll, exploitable via malformed RJS skin (reported by John Heasman) http://www.service.real.com/help/faq/security/041026_player/EN/
Buffer overflow within the RealPlayer ActiveX control (reported by John Heasman) http://www.service.real.com/help/faq/security/040928_player/EN/
Exploitable boundary condition error when handling malformed RM files (reported by John Heasman) http://www.service.real.com/help/faq/security/040610_player/
Buffer overflow handling overly long SMIL tags (reported by Mark Litchfield) http://service.real.com/help/faq/security/security022405.html
Buffer overflow handling malformed R3T files (reported by Mark Litchfield) http://www.service.real.com/help/faq/security/040406_r3t/en/
Buffer overflow handling malformed RM files (reported by Mark Litchfield) http://www.service.real.com/help/faq/security/040123_player/

A quick diversion

- Cross-site scripting in server software…

You Win!

A quick diversion

- Code execution in Internet Explorer via 3rd party ActiveX control

A quick diversion

- Vulnerability in widely used enterprise-level software

A quick diversion

- Remotely exploitability vulnerability in critical infrastructure/the core OS

A quick diversion

- Why isn't everyone in the security community rich…

- The best (= $$$) targets are the most secure
  - Few remote vulnerabilities in Vista
  - Even less in IIS or Apache
  - Clued up vendors are fuzzing:
  - Windows Vista file fuzzing effort in numbers:
    - 350M iterations total
    - 250+ file parsers fuzzed
    - 300+ issues fixed

Advanced fuzzing

- Problems with our basic model:
  - Fuzzer and target are not in sync
  - Fuzzer has no feedback mechanism
  - We have no visibility into fuzzer's effectiveness
- Revised model:
  - Send good requests at regular intervals to check health of target
  - Fuzzer is able to control target debugger
  - Use a code coverage build if possible

Advanced fuzzing

- Fuzzer issues debugger commands
  - Can restart target
  - Can handle resource exhaustion bugs, e.g. 100% CPU
  - Can save callstack, crash dump etc.

- Fuzzing isolated states in a state machine
  - Save and restore “state” of process
  - See Hoglund, McGraw “Exploiting Software”
  - Constrain execution to a single state with breakpoints
  - Substitute real data for fuzzed data
  - But how to handle handles?

Example

- Microsoft Exchange Remote Code Execution
  - Critical Microsoft hotfix, MS06-003 (Heasman & Litchfield)
  - Run code on a mail server with a single email!
  - Bonus: same bug affected Outlook 2003!
  - We found this by fuzzing email…
  - Rich text email often has a “winmail.dat” attachment
  - This is a binary Transport Neutral Encapsulation Format
  - Lets fuzz it!

Example

- TNEF is a Tag Length Value (TLV) format
  - Our bug: triggered by tag length of 0xFFFFFFFF for certain properties
  - Any ideas what was going on?
  - Integer overflow!

```
0xFFFFFFFF + 1 = 0
```

- Small memory allocation followed by a large memcpy
  - We trash the heap, overwriting heap control structures
  - End up with arbitrary DWORD overwrite which we use to get code execution

Example

- Interesting asides: 5 years earlier…
- Bugtraq Security Mailing List - August 2000
  - *"As a side note it would be an interesting excercise to see if Outlook is susceptible to a message with a malformed winmail.dat attached. One could theoretically use winmail.dat to hit on holes in either Outlook itself, or the Outlook RTF engine"*
- February 2009 - MS09-003 “Vulnerabilities in Microsoft Exchange Could Allow Remote Code Execution”
  - TNEF again!

Existing challenges

- Fuzzing likely to trigger same bug repeatedly
  - Automatically remove duplicate bugs
  - Compare call stacks, register values, memory locations
  - What if they are trashed?
  - Or slightly different?
  - Seems like this problem would benefit from a fuzzy approach
- How to implement code coverage on a binary?
  - … Without degrading performance
  - How to effectively use code coverage to direct fuzzing ?

Existing challenges

- How to measure effectiveness of a fuzzer?
  - Number of test cases?
  - Number of bugs?
  - Severity of bugs?
  - % Code coverage?
- How many test cases to run?
  - How to balance complexity vs. time constraints?

Questions?

And one last thing…

NGS is always looking to hire talented people!

john@ngssoftware.com
http://www.ngssoftware.com