Real World Fuzzing

Charlie Miller
Independent Security Evaluators
October 19, 2007

cmiller@securityevaluators.com

Agenda

- Fuzzing 101
- Common Fuzzing Problems
- Code Coverage
- Examples

Fuzzing

- A form of vulnerability analysis and testing
- Many slightly anomalous test cases are input into the target application
- Application is monitored for any sign of error

Example

- Standard HTTP GET request
  - GET /index.html HTTP/1.1
- Anomalous requests
  - AAAAAA...AAAA /index.html HTTP/1.1
  - GET ///////index.html HTTP/1.1
  - GET %n%n%n%n%n%n.html HTTP/1.1
  - GET /AAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1
  - etc...

Different Ways To Fuzz

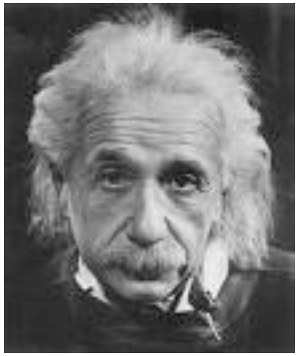
- Mutation Based - “Dumb Fuzzing”
- Generation Based - “Smart Fuzzing”
- Evolutionary

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics
- Requires little to no set up time
- Dependent on the inputs being modified
- May fail for protocols with checksums, those which depend on challenge response, etc.
- Examples:
  - Taof, GPF, ProxyFuzz, etc.

Generation Based Fuzzing

- Test cases are generated from some description of the format: RFC, documentation, etc.
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing
- Can take significant time to set up
- Examples
  - SPIKE, Sulley, Mu-4000, Codenomicon



Evolutionary Fuzzing

- Attempts to generate inputs based on the response of the program
- Autodafe
  - Prioritizes test cases based on which inputs have reached dangerous API functions
- EFS
  - Generates test cases based on code coverage metrics (more later)
- This technique is still in the alpha stage :)

The Problems With Fuzzing

- Mutation based fuzzers can generate an infinite number of test cases... When has the fuzzer run long enough?
- Generation based fuzzers generate a finite number of test cases. What happens when they're all run and no bugs are found?
- How do you monitor the target application such that you know when something "bad" has happened?

The Problems With Fuzzing

- What happens when you find too many bugs? Or every anomalous test case triggers the same (boring) bug?
- How do you figure out which test case caused the fault?
- Given a crash, how do you find the actual vulnerability
- After fuzzing, how do you know what changes to make to improve your fuzzer?
- When do you give up on fuzzing an application?

Example 1: PDF

- Have a PDF file with 248,000 bytes
- There is one byte that, if changed to particular values, causes a crash
  - This byte is 94% of the way through the file
- Any single random mutation to the file has a probability of .00000392 of finding the crash
- On average, need 127,512 test cases to find it
- At 2 seconds a test case, thats just under 3 days...
- It could take a week or more...

Example 2: 3g2

- Video file format
- Changing a byte in the file to 0xff crashes QuickTime Player 42% of the time
- All these crashes seem to be from the same bug
- There may be other bugs “hidden” by this bug
- FIXED SILENTLY LAST WEEK - NEVER MIND!

Code Coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric which can be used to determine how much code has been executed.
- Works for source code or binaries, although almost all the literature assumes you have source

Line Coverage

- Measures how many lines of code (source code lines or assembly instructions) have been executed.

Branch Coverage

- Measures how many branches in code have been taken (conditional jmps)

```
if( x > 2 )
  x = 2;
```

- The above code can achieve full line coverage in one test case (ex. `x=3`)
- Requires 2 test cases for total branch coverage (ex. `x=1, x=2`).

Path Coverage

- Measures the number of paths executed

```
if( a > 2 )
  a = 2;
if( b > 2 )
  b = 2;
```

- Requires
  - 1 test case for line coverage
  - 2 test cases for branch coverage
  - 4 test cases for path coverage
    - i.e. `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`

Path Coverage Issues

- In general, a program with `n` "reachable" branches will require `2n` test cases for branch coverage and `2^n` test cases for path coverage
  - § Umm....there's a lot of paths in a program!
- If you consider loops, there are an infinite number of paths
- Some paths are *infeasible*

```
if(x>2)
   x=2;
if(x<0)
   x=0;
```

  - § You can't satisfy both of these conditionals, i.e. there is only three paths through this code, not four

Getting Code Coverage Data

- If you've got source
  - Instrument the code while compiling
    - gcov
    - Insure++
    - Bullseye

Getting Code Coverage Data

- If you live in the real world
  - Use Debugging info
    - Pai Mei
  - Virtualization
    - Valgrind
    - Bochs
    - Xen?
  - Dynamic code instrumentation
    - DynamoRIO
    - Aprobe

Problems with Code Coverage

- Code can be covered without revealing bugs

```
mySafeCpy(char *dst, char* src){
     if(dst && src)
          strcpy(dst, src);
}
```

- Error checking code mostly missed (and we don't particularly care about it)

```
ptr = malloc(sizeof(blah));
if(!ptr)
     ran_out_of_memory();
```

- Only "attack surface" reachable
  - i.e. the code processing user controlled data
  - No easy way to measure the attack surface

Now the Examples

- Note: we start with some source code examples but move on to binary only

The Hello World of Code Coverage

- Simple program with 3 paths

```
int main(int argc, char *argv[]){
        if(argc == 2){
                if(strstr(argv[1], "hi")){
                        printf(" Hello world\n");
                }
        } else {
                printf("Wrong number of arguments\n");
        }
        return 1;
}
```

Gcov

- Compile with "coverage" flags

```
gcc -g -fprofile-arcs -ftest-coverage -o hello hello.c
```

- This generates a .gcno file for each object file which contains static information about it, such as locations of branches, names of functions, etc

Under the Hood

```
0x00001b0a <main+0>:    push   ebp
0x00001b0b <main+1>:    mov    ebp,esp
0x00001b0d <main+3>:    push   ebx
0x00001b0e <main+4>:    sub    esp,0x14
0x00001b11 <main+7>:    call   0x2ffc <__i686.get_pc_thunk.bx>
0x00001b16 <main+12>:   cmp    DWORD PTR [ebp+8],0x2
0x00001b1a <main+16>:   jne    0x1b77 <main+109>
0x00001b1c <main+18>:   lea    eax,[ebx+0x158a]
0x00001b22 <main+24>:   add    DWORD PTR [eax],0x1
0x00001b25 <main+27>:   adc    DWORD PTR [eax+4],0x0
```

- Additional code added to binary
- 64-bit global variable stores coverage information
- Dumped to disk when gcov_exit() is called

Run it

```
$ ./hello there
$ ./hello hi_there
 Hello world
```

- When you run the program, code coverage information is stored in .gcda files for each object file
- To process these files, run gcov

```
$ gcov hello.c
File 'hello.c'
Lines executed:83.33% of 6
hello.c:creating 'hello.c.gcov'
```

hello.c.gcov

```
        -:    0:Source:hello.c
        -:    0:Graph:hello.gcno
        -:    0:Data:hello.gcda
        -:    0:Runs:2
        -:    0:Programs:1
        2:    1:int main(int argc, char *argv[]){
        2:    2:        if(argc == 2){
        2:    3:                if(strstr(argv[1], "hi")){
        1:    4:                        printf(" Hello world\n");
        -:    5:                }
        -:    6:        } else {
    #####:    7:                printf("Wrong number of arguments\n");
        -:    8:        }
        2:    9:        return 1;
        -:   10:}
```

In June 2007...

- A group of cunning, good looking researchers hacked the iPhone
- How'd we find the bug?

- Fuzzing + Code Coverage!

WebKit

- Most Apple Internet applications share the same code, WebKit
- WebKit is an open source library
- Source code is available via svn:
  - svn checkout http://svn.webkit.org/repository/webkit/trunk WebKit

Thanks

- From the development site:

The JavaScriptCore Tests

If you are making changes to JavaScriptCore, there is an additional test suite you must run before landing changes. This is the Mozilla JavaScript test suite.

- So we know what they use for unit testing
- Let’s use code coverage to see which portions of code might not be as well tested

lcov

- One problem with gcov is the data is stored in many different files
- lcov is an open source software package which collects data from a whole project and displays it in a nice html report
- It can be a minor pain in the ass to get to work...

Build and Run WebKit

- Build it:

```
WebKit/WebKitTools/Scripts/build-webkit -coverage
```

- Run the test suite:

```
WebKitTools/Scripts/run-javascriptcore-tests -coverage
```

- Add a bunch of stupid links for lcov...sigh :(
- Collect coverage data

```
lcov --directory WebKitBuild/JavaScriptCore.build/Release/
JavaScriptCore.build/Objects-normal/i386 -c -o testsuite.info
```

- Generate HTML report

```
genhtml -o WebKit-html -f testsuite.info
```

Results

LTP GCOV extension - code coverage report

Current view:	directory		
Test:	testsuite.info		
Date:	2007-06-01	Instrumented lines:	13622
Code covered:	59.3 %	Executed lines:	8073

Directory name	Coverage		
/System/Library/Frameworks/CoreFoundation.framework/Headers		100.0 %	1 / 1 lines
/System/Library/Frameworks/JavaVM.framework/Headers		0.0 %	0 / 53 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/API		0.0 %	0 / 474 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings		0.0 %	0 / 530 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/c		0.0 %	0 / 190 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/jni		0.0 %	0 / 890 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/bindings/objc		0.0 %	0 / 476 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/kjs		79.3 %	5723 / 7219 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/pcre		54.7 %	1338 / 2445 lines
/Users/cmiller/woot/WebKit/JavaScriptCore/wtf		0.0 %	0 / 56 lines
/usr/include		100.0 %	2 / 2 lines
/usr/include/architecture/i386		100.0 %	3 / 3 lines
/usr/include/c++/4.0.0/bits		50.0 %	4 / 8 lines
/usr/share		89.7 %	96 / 107 lines
JavaScriptCore/kjs		84.8 %	357 / 421 lines
kjs		0.0 %	0 / 39 lines
wtf		76.9 %	528 / 687 lines
wtf/unicode/icu		100.0 %	21 / 21 lines

Generated by: LTP GCOV extension version 1.5

Results

- 59.3% of 13622 lines in JavaScriptCore were covered
  - The main engine (53% of the overall code) had 79.3% of its lines covered
  - Perl Compatible Regular Expression (PCRE) library (17% of the overall code) had 54.7% of its lines covered
- We decided to investigate PCRE further

...The Rest of the Story

- Wrote a PCRE fuzzer (20 lines of perl)
- Ran it on a standalone PCRE parser (pcredemo from the PCRE library)
- We started getting errors like:

```
PCRE compilation failed at offset 6: internal error: code overflow.
```

- This was good

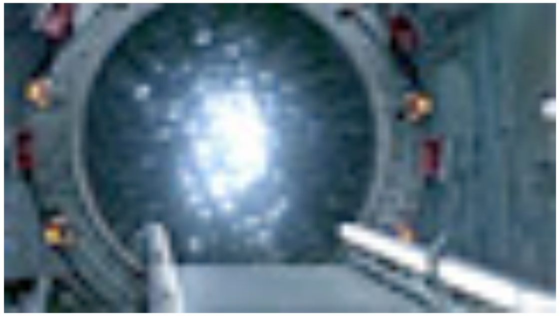
A Short Digression on iPhone Hacking: - or - How To Write an Exploit by Fuzzing

- Using our evil regular expression, we could crash mobileSafari (which uses Webkit)
- We didn't have a debugger for the iPhone.
- We couldn't compile code for the iPhone
- We did have crash reports which gave register values
- We did have core dumps (after some iPhone modifications)

All Exploits Need...

- To get control (in this case `pc = r15`)
- To find your shellcode

- Q: How can you do this without a debugger?
- A: The same way you find bugs while watching TV: fuzzing



Fuzz to Exploit

- We generated hundreds of regular expressions containing different number of “evil” strings: “[[**]]”
- Sorted through the crash reports
- Eventually found a good one

A "Good" Crash

```
Thread 2 crashed with ARM Thread State:
r0: 0x00065000    r1: 0x0084f800     r2: 0x00000017     r3: 0x15621561
r4: 0x00000018    r5: 0x0084ee00     r6: 0x00065000     r7: 0x005523ac
r8: 0x0000afaf    r9: 0x00817a00    r10: 0x00ff8000    r11: 0x00000005
ip: 0x15641563    sp: 0x00552358     lr: 0x30003d70     pc: 0x3008cbc4
cpsr: 0x20000010 instr: 0xe583c004
```

```
__text:3008CBC4                 STR     R12, [R3,#4]
__text:3008CBC8                 BXEQ    LR
__text:3008CBCC
__text:3008CBCC loc_3008CBCC                            ; CODE XREF: __text:3008CBA0j
__text:3008CBCC                 STR     R3, [R12]
```

- Unlinking of a linked list
- `r3` and `r12`=ip are controllable
- Old school heap overflow (gotta love Apple)
- Gives us a "write anywhere" primitive
- Hows it work? Who the hell knows!
- HD Moore, who is an exploit writing genius, would be sad :(

More Fuzzing For Exploitation

- We decided to overwrite a return address on the stack.
- How do you find it? Fuzz!
  - True fuzzing folks will call this brute forcing and not fuzzing, but either way its easy...

```
Exception Type:  EXC_BAD_INSTRUCTION
...
Thread 2 crashed with ARM Thread State:
    r0: 0x00065038    r1: 0x00000000      r2: 0x00000a00      r3: 0x00000001
    r4: 0x00065000    r5: 0x380135a4      r6: 0x00000000      r7: 0x005523e4
    r8: 0x00000000    r9: 0x00815a00     r10: 0x0084b800     r11: 0x00000000
    ip: 0x380075fc    sp: 0x005523d0      lr: 0x30003e18      pc: 0x0055ff3c
  cpsr: 0x20000010 instr: 0xffffffff
```

PNG - with source

- libpng-1.2.16
- Used in Firefox, Safari, and Thunderbird (and others)
- http://www.libpng.org/pub/png/libpng.html

Build the Source

- ./configure CFLAGS="-g -fprofile-arcs -ftest-coverage"
- make (errors out)
- gcc -g -fprofile-arcs -ftest-coverage -I. -L/usr/X11R6/lib/ -I/usr/X11R6/include contrib/gregbook/rpng-x.c .libs/libpng12_la-png.o .libs/libpng12_la-pngset.o .libs/libpng12_la-pngget.o .libs/libpng12_la-pngrutil.o .libs/libpng12_la-pngtrans.o .libs/libpng12_la-pngwutil.o .libs/libpng12_la-pngread.o .libs/libpng12_la-pngrio.o .libs/libpng12_la-pngwio.o .libs/libpng12_la-pngwrite.o .libs/libpng12_la-pngrtran.o .libs/libpng12_la-pngwtran.o .libs/libpng12_la-pngmem.o .libs/libpng12_la-pngerror.o .libs/libpng12_la-pngpread.o .libs/libpng12_la-pnggccrd.o contrib/gregbook/readpng.c -o contrib/gregbook/rpng-x -lX11 -lz -lgcov
- result: contrib/gregbook/rpng-x

Quick Test

```
$ ./contrib/gregbook/rpng-x
$ find . | grep gcda
./.libs/libpng12_la-png.gcda
./.libs/libpng12_la-pngerror.gcda
./.libs/libpng12_la-pnggccrd.gcda
./.libs/libpng12_la-pngget.gcda
./.libs/libpng12_la-pngmem.gcda
./.libs/libpng12_la-pngpread.gcda
…
```

How ‘bout a Little Dumb Fuzzing Action?

- Grab a PNG off the Internet
  - The first one I find is from Wikipedia: PNG_transparency_demonstration_1.png
- Zero out any code coverage data
  - `lcov --directory . -z`

Generate Some Files

- Use fuzz.c, the “super” fuzzer
  - Changes 1-17 bytes in each file
  - New value is random
  - Does this 8192 times
- The ultimate in dumb fuzzer technology

```
./fuzz > fuzz.out
```

Use the Files

- Use script.sh
  - Executes the program 10 at a time
  - Sleeps 5 seconds
  - Kills any processes
  - Repeats
  - Monitors CrashReporter log for crashes

Get Code Coverage

- We covered 10.7% of the lines

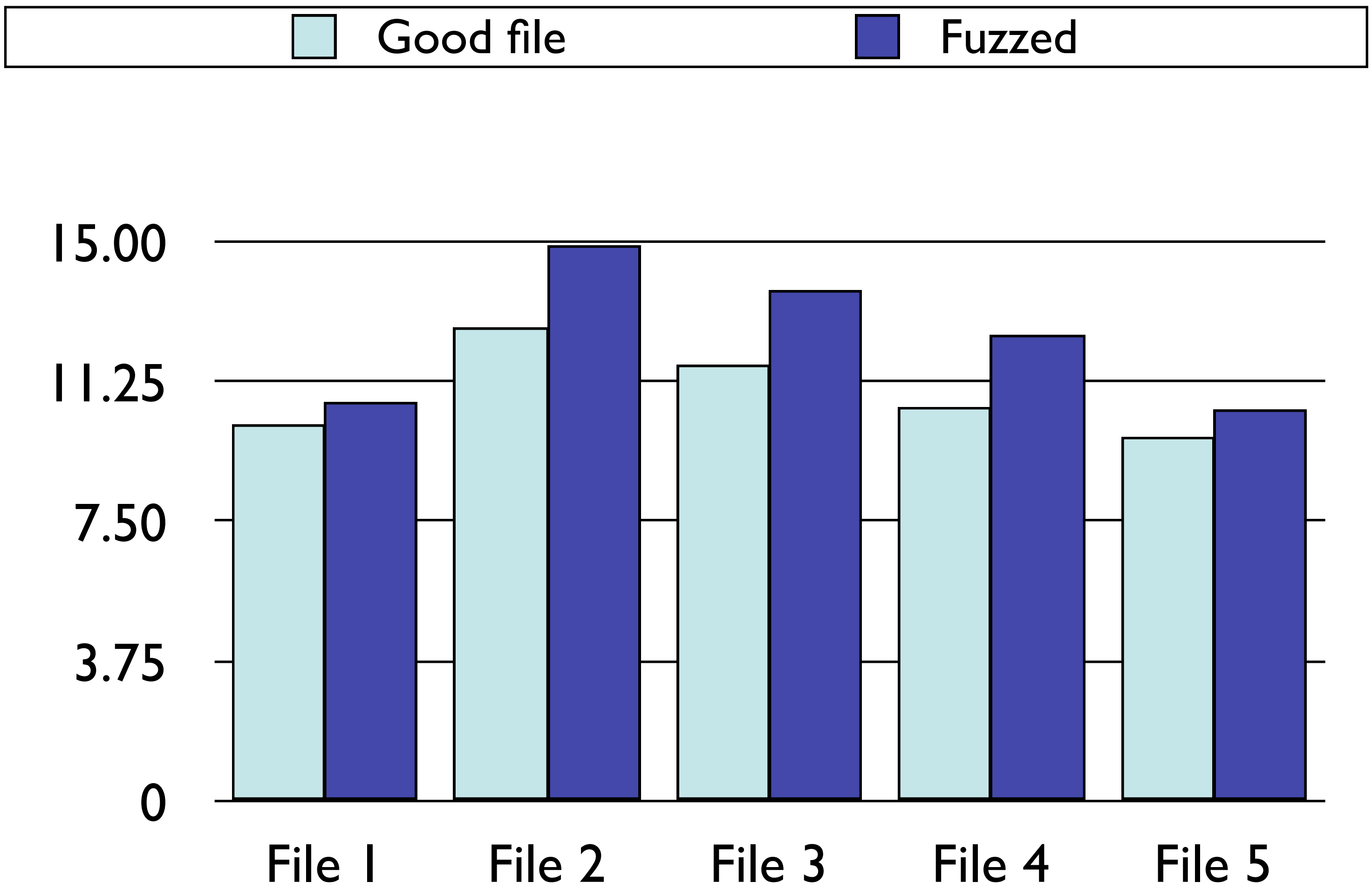
```
cp *.c .libs/
lcov --directory . -c -o fuzz.info
genhtml -f -o fuzz_html_files fuzz.info
...
```

- This compares to
  - 0.4% for getting the usage statement
  - 745 of 7399 (10.1%) for opening the good file
    - 43 more lines covered by fuzzing...

What’s Up?

- That code coverage kinda sucked...
- Did we choose a bad initial file
- Let’s try some other files...
  - Choose 4 other PNG’s from the Internet
  - Fuzz them the same way
  - Collect data from each separately

Results

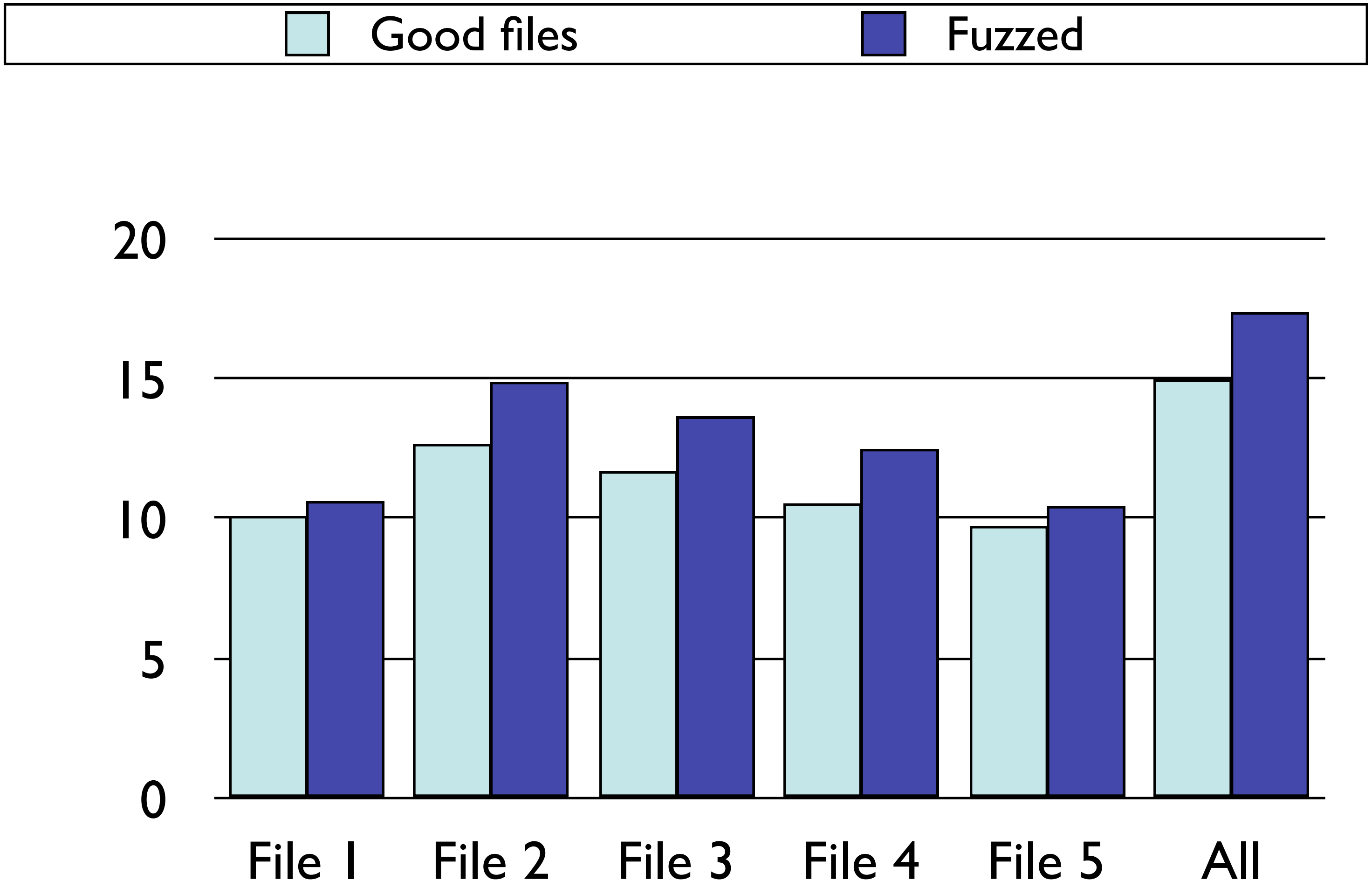


Good file
Fuzzed
15.00
11.25
7.50
3.75
0
File 1
File 2
File 3
File 4
File 5

So...

- Initial file can make a big difference
  - 50% more code coverage from file 2 than in file 5
- What if we ran them all?

The Sum is Greater Than the Parts



WTF is Going On?

- Each PNG contains certain elements that requires some code to process
- Some PNG's contain the same elements, some contain different ones
- By fuzzing with a variety of different PNG's, you increase the chance of having different elements which need processing
- Charlie's Heuristic: Keep adding files until the cumulative effect doesn't increase

A Brief Interlude Into PNG's

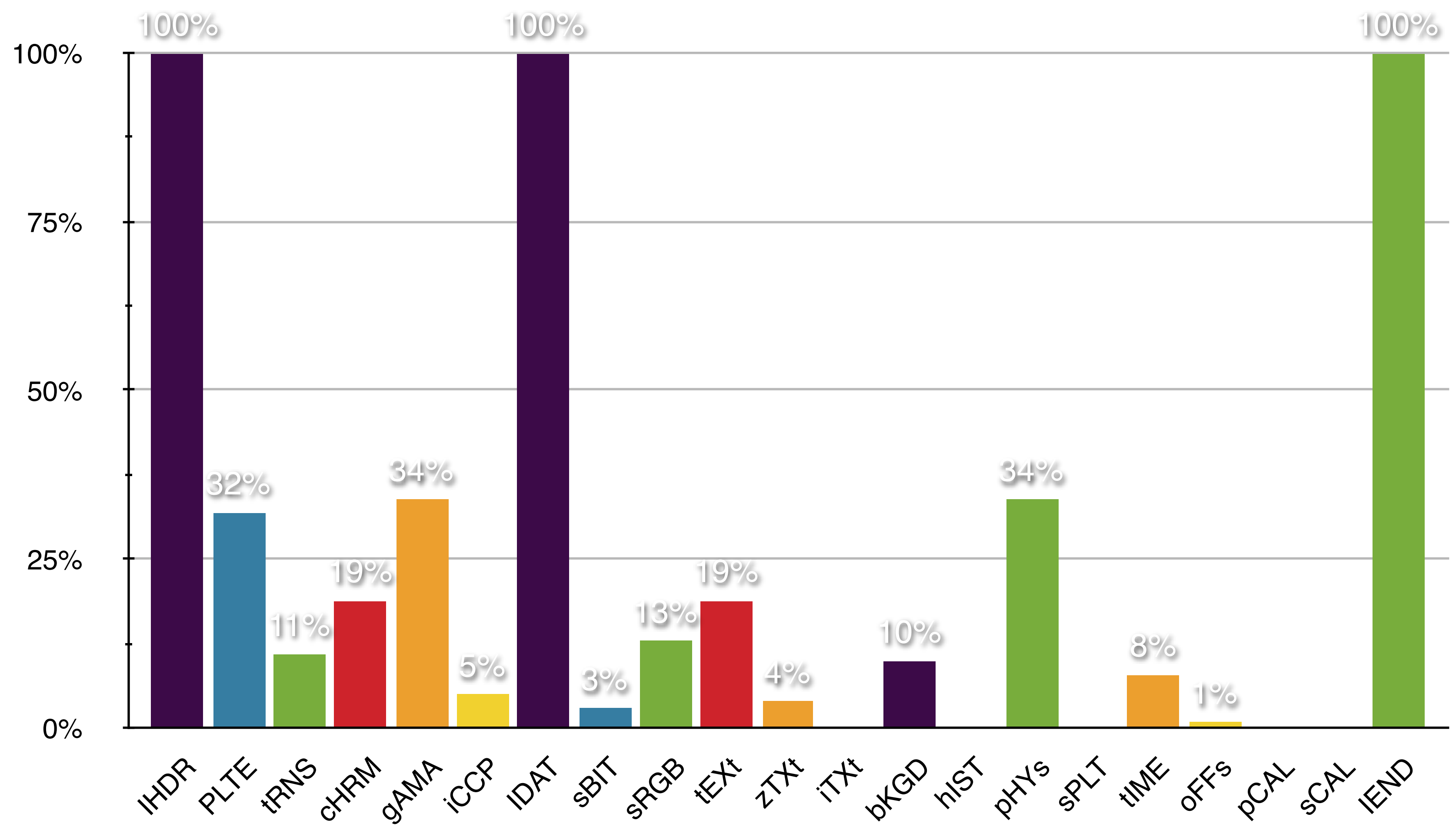
- 8 byte signature followed by "chunks"
- Each chunk has
  - 4 byte length field
  - 4 byte type field
  - optional data
  - 4 byte CRC checksum
- 18 chunk types, 3 of which are mandatory
- Additional types are defined in extensions to the specification
  - libpng supports 21 chunk types

PNG's From the Wild

- Collected 1631 unique PNG files from the Internet
- Each file was processed and the chunk types present in each was recorded
- Typically, very few chunk types were present

Number of files	Mean number of chunk types	Standard deviation	Maximum	Minimum
1631	4.9	1.3	9	3

Distribution of Chunks Found



Observations

- On average, only five of the chunk types are present in a random file!
- 9 of the 21 types occurred in less than 5% of files sampled
- 4 of the chunk types never occurred
- Mutation based fuzzers will typically only test the code from these five chunks
- *They will never fuzz the code in chunks which are not present in the original input*

Enter Generation-Based Fuzzers

- Since Generation-based fuzzers build test cases not from valid data, but from the specification, they should contain all possible chunks
- This should make for a more thorough test

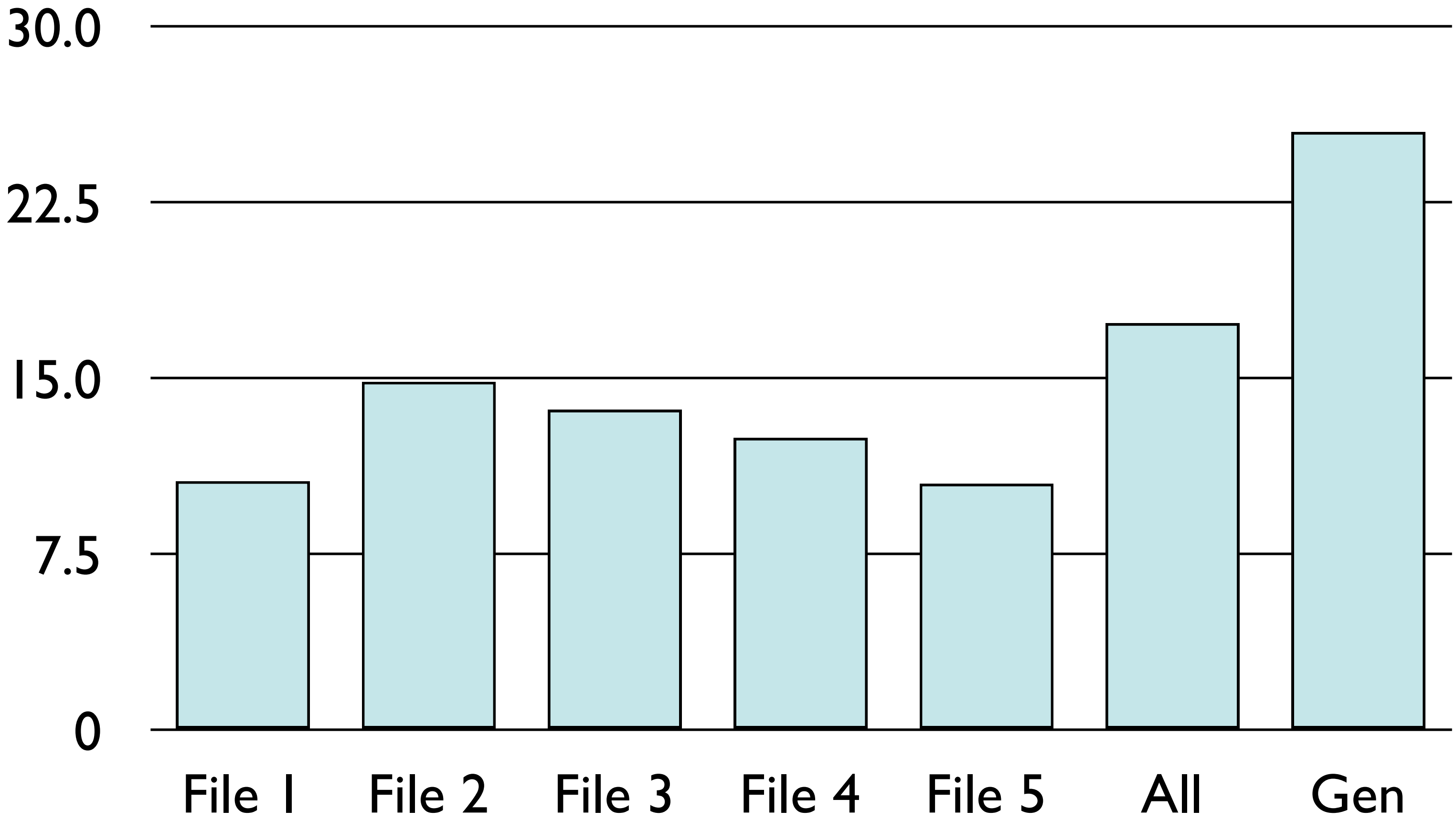
SPIKE

```
//png.spk
// Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based 
on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation Gap



The Point Is

- Questions such as
  - How many initial inputs should I use during mutation based fuzzing?
  - How much effort am I willing to expend to do generational based fuzzing?
- can be answered with code coverage

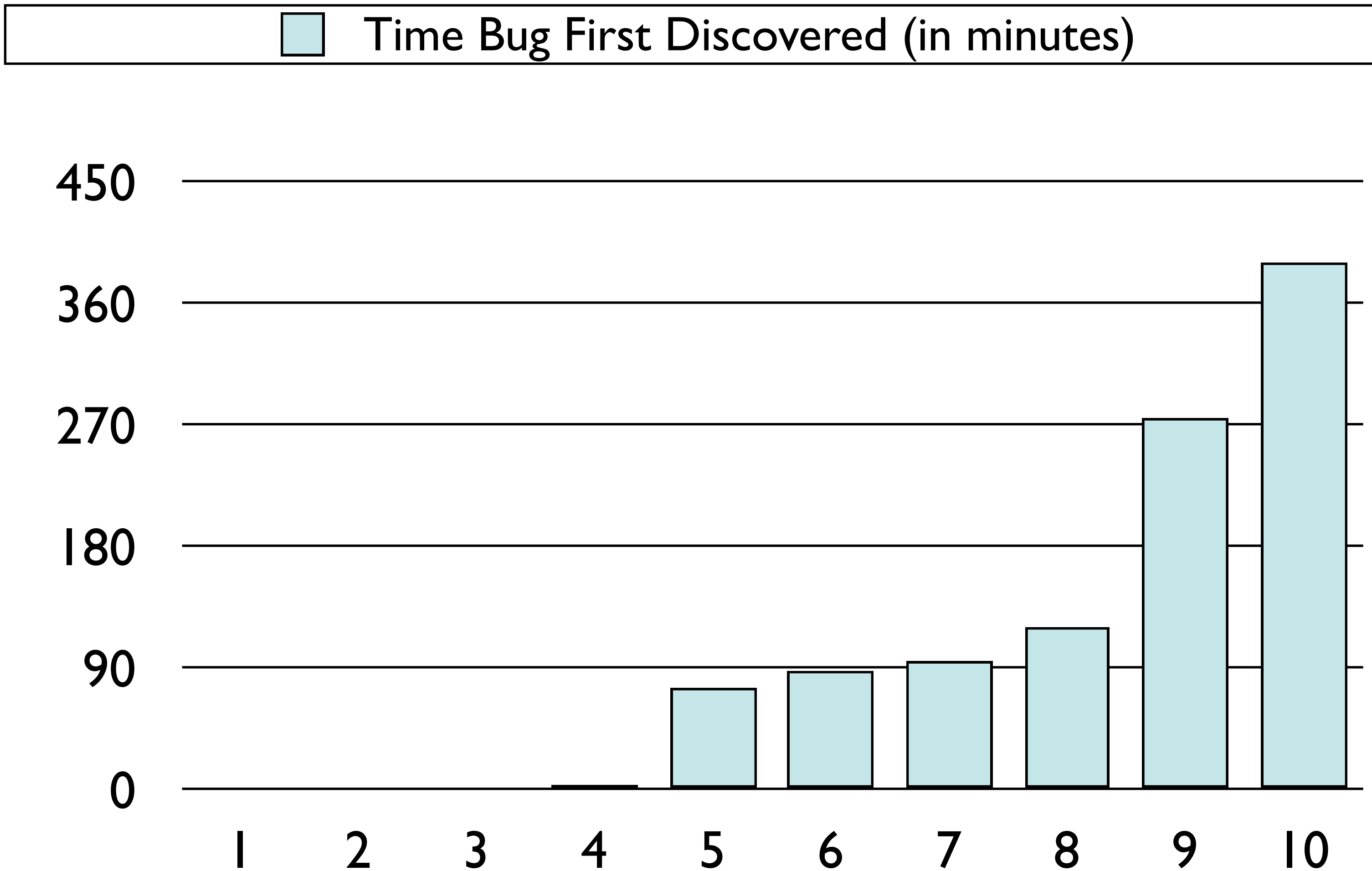
Halting Problem (Again)

- During all this testing
  - Used mutation and generation based fuzzers
  - Generated over 200,000 test cases
  - ***Not one crash***
- This is a common occurrence for difficult or well audited target applications
- Raises the question: Now what?
- Answer later...
  - (Hint: has to do with code coverage)

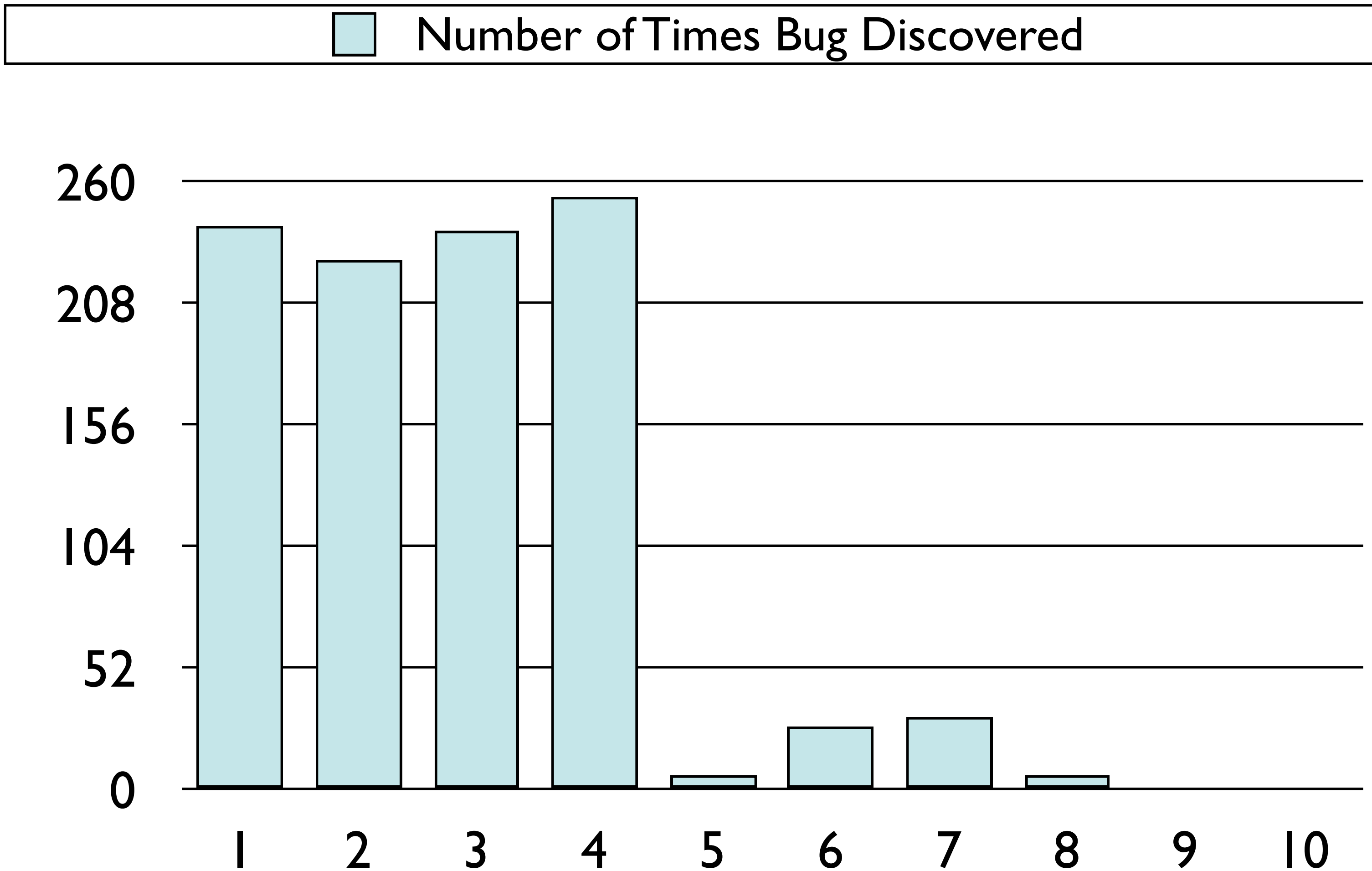
Even More Halting Problem…

- Added 20 “fake” bugs to a server
- Ran ProxyFuzz, a mutation-based fuzzer against it for 450 minutes
- Recorded when each bug was found and how often
- Can you “guess” when to turn off your dumb fuzzer?

Time Required To Find a Bug



Number of Times Discovered



Results

- Sometimes you find a “rare” bug earlier than an “easy” bug
- There are discrete jumps in the time between finding bugs
  - § 4 bugs found in the first 3 minutes
    - Then it took 76 minutes to find the next one
  - § 8 bugs found in the first 121 minutes
    - Then it took another 155 minutes to find the next one
- The final hour didn’t find a new bug, what if I would have run it another 24 hours?

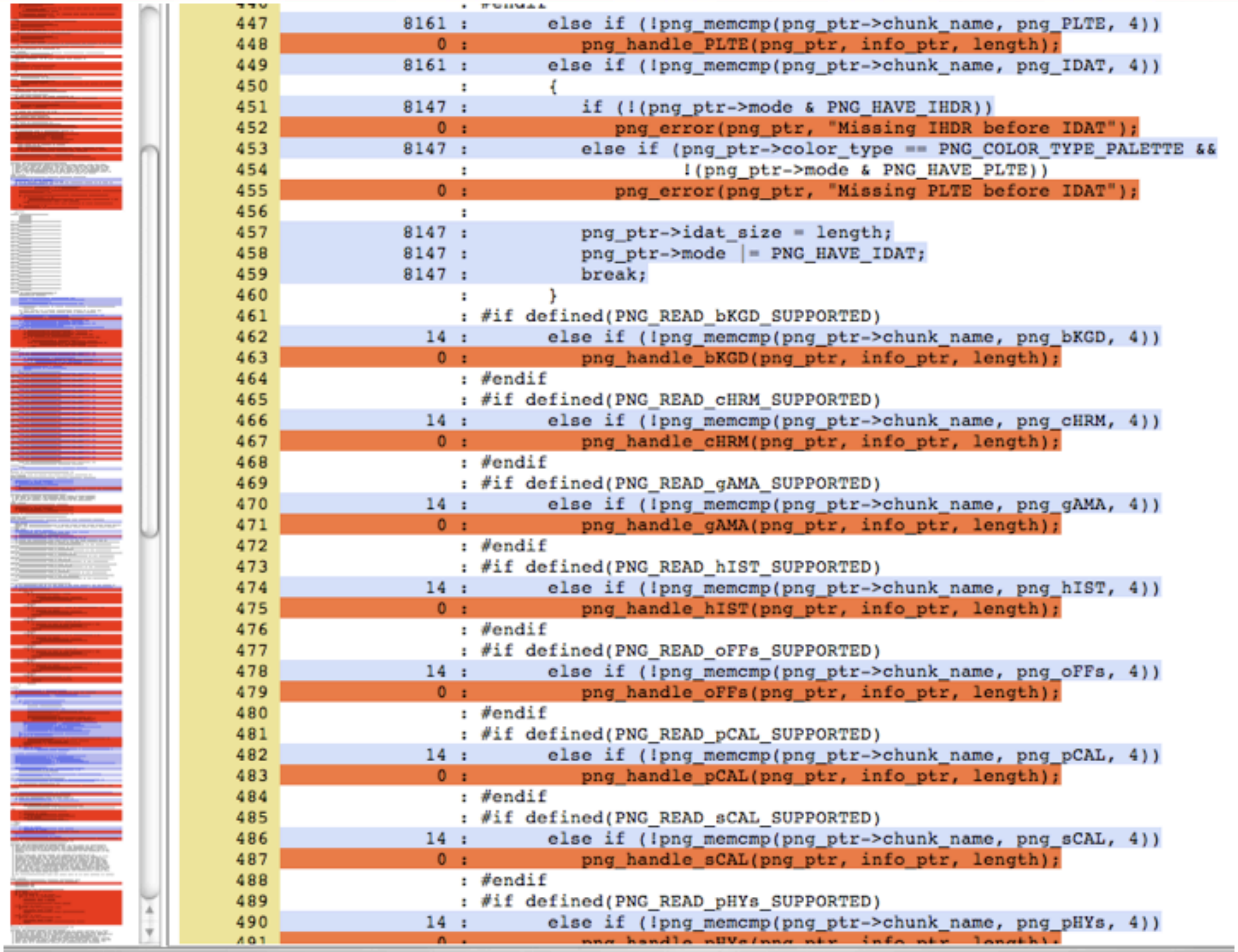
Code Coverage is…

- We've seen that code coverage is
  - A metric to find results about fuzzing
  - Helpful in figuring out general approaches to fuzzing
  - Useful to find what code to focus fuzzing upon
- More importantly (we'll see):
  - A way to improve fuzzing and find more bugs!
  - Helpful in figuring out when fuzzing is "finished"

Look

- Suppose we didn't know anything about PNG's
- Could we have figured out what was missing when we were fuzzing PNG with the mutation based approach?
- Lets look through some of the lcov report

Yup



```
              : #endif
         8161 :         else if (!png_memcmp(png_ptr->chunk_name, png_PLTE, 4))
            0 :            png_handle_PLTE(png_ptr, info_ptr, length);
         8161 :         else if (!png_memcmp(png_ptr->chunk_name, png_IDAT, 4))
              :         {
         8147 :            if (!(png_ptr->mode & PNG_HAVE_IHDR))
            0 :               png_error(png_ptr, "Missing IHDR before IDAT");
         8147 :            else if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE &&
              :                     !(png_ptr->mode & PNG_HAVE_PLTE))
            0 :               png_error(png_ptr, "Missing PLTE before IDAT");
              :
         8147 :            png_ptr->idat_size = length;
         8147 :            png_ptr->mode |= PNG_HAVE_IDAT;
         8147 :            break;
              :         }
              : #if defined(PNG_READ_bKGD_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_bKGD, 4))
            0 :            png_handle_bKGD(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_cHRM_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_cHRM, 4))
            0 :            png_handle_cHRM(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_gAMA_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_gAMA, 4))
            0 :            png_handle_gAMA(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_hIST_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_hIST, 4))
            0 :            png_handle_hIST(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_oFFs_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_oFFs, 4))
            0 :            png_handle_oFFs(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_pCAL_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_pCAL, 4))
            0 :            png_handle_pCAL(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_sCAL_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_sCAL, 4))
            0 :            png_handle_sCAL(png_ptr, info_ptr, length);
              : #endif
              : #if defined(PNG_READ_pHYs_SUPPORTED)
           14 :         else if (!png_memcmp(png_ptr->chunk_name, png_pHYs, 4))
            0 :            png_handle_pHYs(png_ptr, info_ptr, length);
```

Code Coverage Improves Fuzzing

- Finding spots in the code which are not covered can help with the generation of new test cases
- Beware: covered code doesn't necessarily mean its "fuzzed"
- *Code which has not been executed definitely still needs to be fuzzed!*

Digression into Binary Code Coverage

- So far, we've seen how code coverage can give useful information to help fuzzing
- We've seen how to use gcov and lcov to do this
- The exact same data can be obtained on Windows binaries using Pai Mei
- Pai Mei exists for Mac OS X and is being ported to Linux

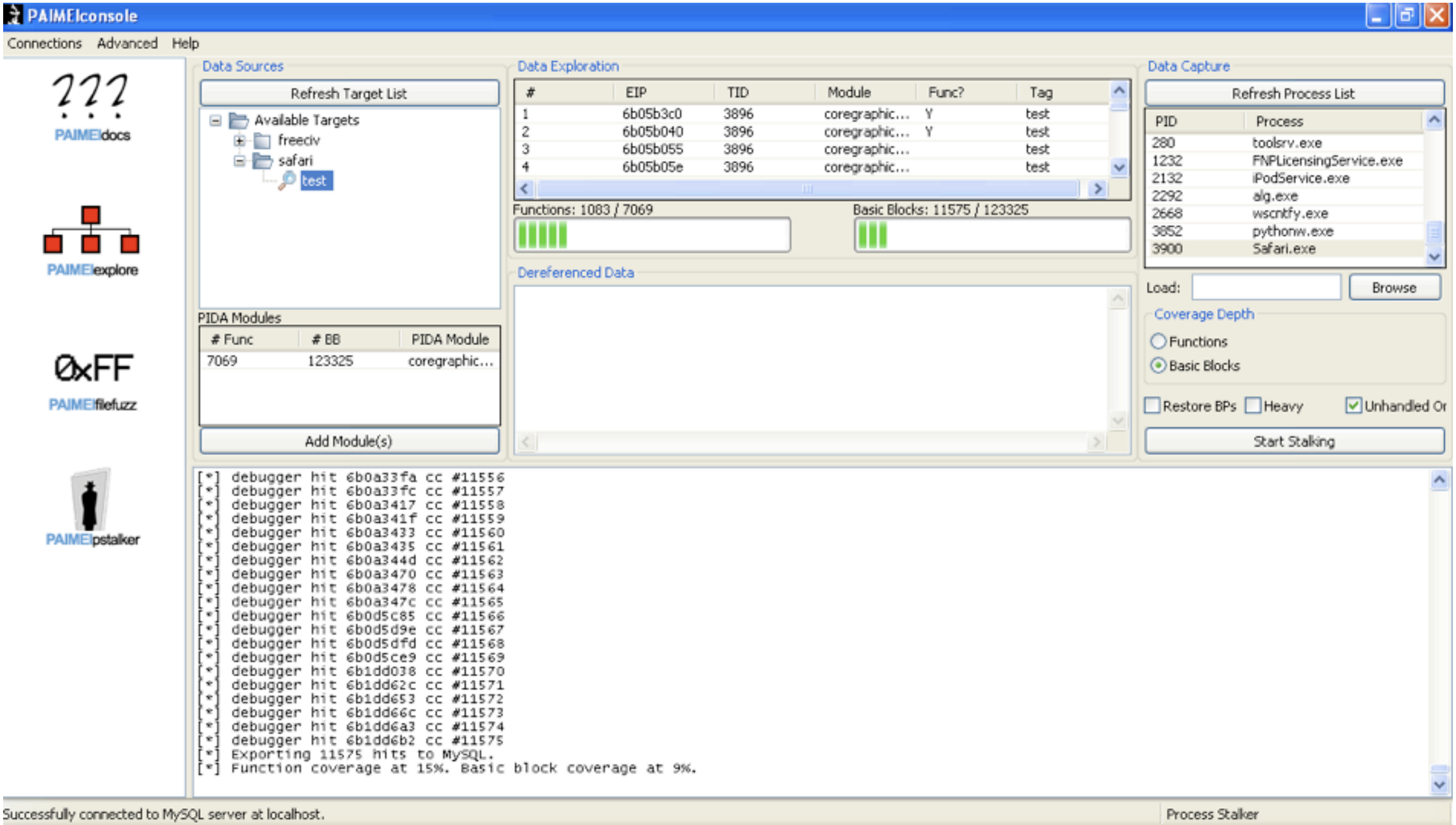
Pai Mei

- A reverse engineering framework
- Integrates
  - PyDbg debugger
  - IDA Pro databases (via PIDA)
  - pGraph graphing
  - mySQL database
- Gives the ability to perform reverse engineering tasks quickly and repeatably
- http://paimei.openrce.org/

Pstalker

- A Pai Mei Module
- Uses IDA Pro to get structure of binary
- Sets breakpoints at each basic block (or function)
- Records and removes breakpoints that are hit
- Allows for filtering of breakpoints
- *Gathers code coverage for binaries*

Screenshot



PAIMEIconsole
Connections Advanced Help
PAIMEIdocs
PAIMEIexplore
0xFF
PAIMEIfilefuzz
PAIMEIpstalker
Data Sources
Refresh Target List
Available Targets
freeciv
safari
test
PIDA Modules
Func # BB PIDA Module
7069 123325 coregraphic...
Add Module(s)
Data Exploration
EIP TID Module Func? Tag
1 6b05b3c0 3896 coregraphic... Y test
2 6b05b040 3896 coregraphic... Y test
3 6b05b055 3896 coregraphic... test
4 6b05b05e 3896 coregraphic... test
Functions: 1083 / 7069
Basic Blocks: 11575 / 123325
Dereferenced Data
Data Capture
Refresh Process List
PID Process
280 toolsrv.exe
1232 FNPLicensingService.exe
2132 iPodService.exe
2292 alg.exe
2668 wscntfy.exe
3852 pythonw.exe
3900 Safari.exe
Load:
Browse
Coverage Depth
Functions
Basic Blocks
Restore BPs
Heavy
Unhandled Or
Start Stalking
[*] debugger hit 6b0a33fa cc #11556
[*] debugger hit 6b0a33fc cc #11557
[*] debugger hit 6b0a3417 cc #11558
[*] debugger hit 6b0a341f cc #11559
[*] debugger hit 6b0a3433 cc #11560
[*] debugger hit 6b0a3435 cc #11561
[*] debugger hit 6b0a344d cc #11562
[*] debugger hit 6b0a3470 cc #11563
[*] debugger hit 6b0a3478 cc #11564
[*] debugger hit 6b0a347c cc #11565
[*] debugger hit 6b0d5c85 cc #11566
[*] debugger hit 6b0d5d9e cc #11567
[*] debugger hit 6b0d5dfd cc #11568
[*] debugger hit 6b0d5ce9 cc #11569
[*] debugger hit 6b1dd038 cc #11570
[*] debugger hit 6b1dd62c cc #11571
[*] debugger hit 6b1dd653 cc #11572
[*] debugger hit 6b1dd66c cc #11573
[*] debugger hit 6b1dd6a3 cc #11574
[*] debugger hit 6b1dd6b2 cc #11575
[*] Exporting 11575 hits to MySQL.
[*] Function coverage at 15%. Basic block coverage at 9%.
Successfully connected to MySQL server at localhost.
Process Stalker

One Hitch

- Can’t keep launching the process
- Have to have a way for it to keep loading the fuzzed images
- Just use a meta-refresh tag and point the browser at it

The Fuzzing Website

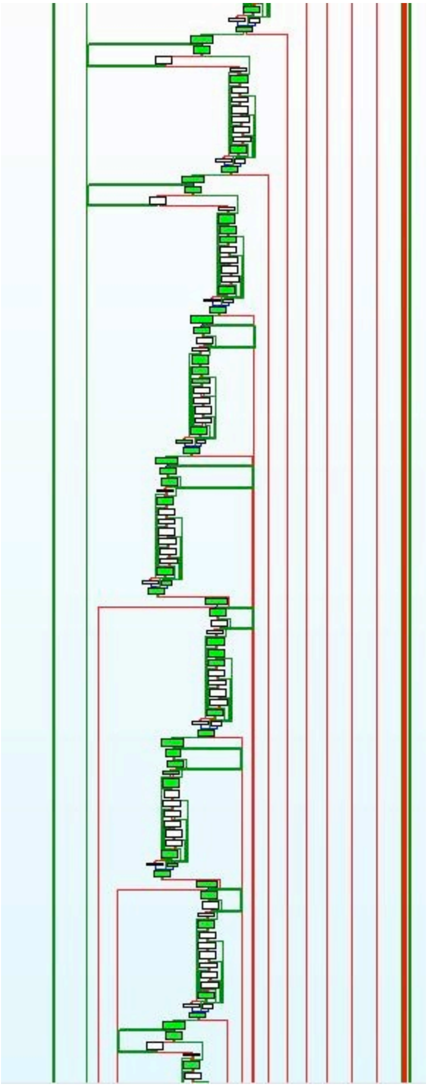
```
#!/usr/bin/perl

$file = $ENV{'QUERY_STRING'};
$nextfile = $file + 1;
$server = $ENV{'SERVER_NAME'};
$script = $ENV{'SCRIPT_NAME'};
$url = "http://".$server.$script."?".$nextfile;
$pic = sprintf("bad-%d.gif", $nextfile);
$picurl = "http://".$server."/gif/".$pic;

print "Content-type: text/html

<head>
        Fuzz!
";
print " <meta http-equiv=\"refresh\" content=\"2;$url\">";
print " </head><body>";
print"</body>\n";
print "<Script Language=\"JavaScript\">";
print "window.open('$picurl');";
print "</Script>";
```

Missed PNG Basic Blocks



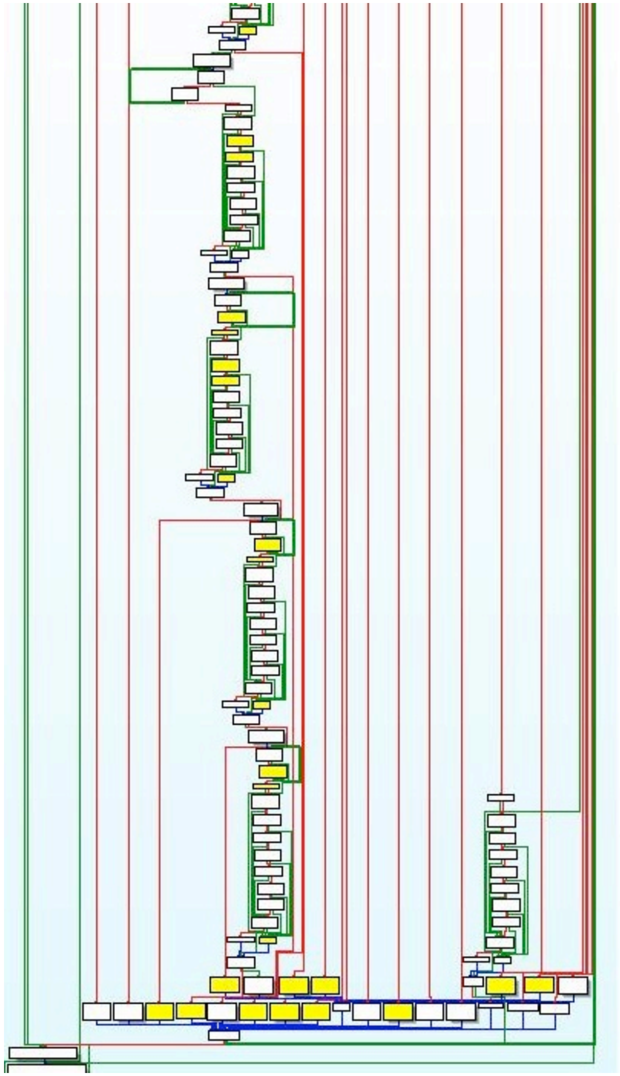
Using Pai Mei to Find the Code

- Do some general browsing in Safari under Pai Mei
  - Avoid pages with PNG's if possible
  - Stop when no more breakpoints are hit
- Record this code coverage in a tag
- Filter out on that tag and browse a bunch of different PNG's
- This will record those basic blocks used only in PNG processing (mostly)

This Results In:

- Total basic blocks: 123,325
- Hit during “general browsing”: 12,776
- Hit during PNG only surfing with filter on: 1094 (0.9% of total basic blocks)
  - This includes 87 functions (out of 7069)
  - 61 of these basic blocks are in the “main” PNG processing function
  - Most of the others are in “chunk” specific functions

Where’s the Code?



Pai Mei Limitations

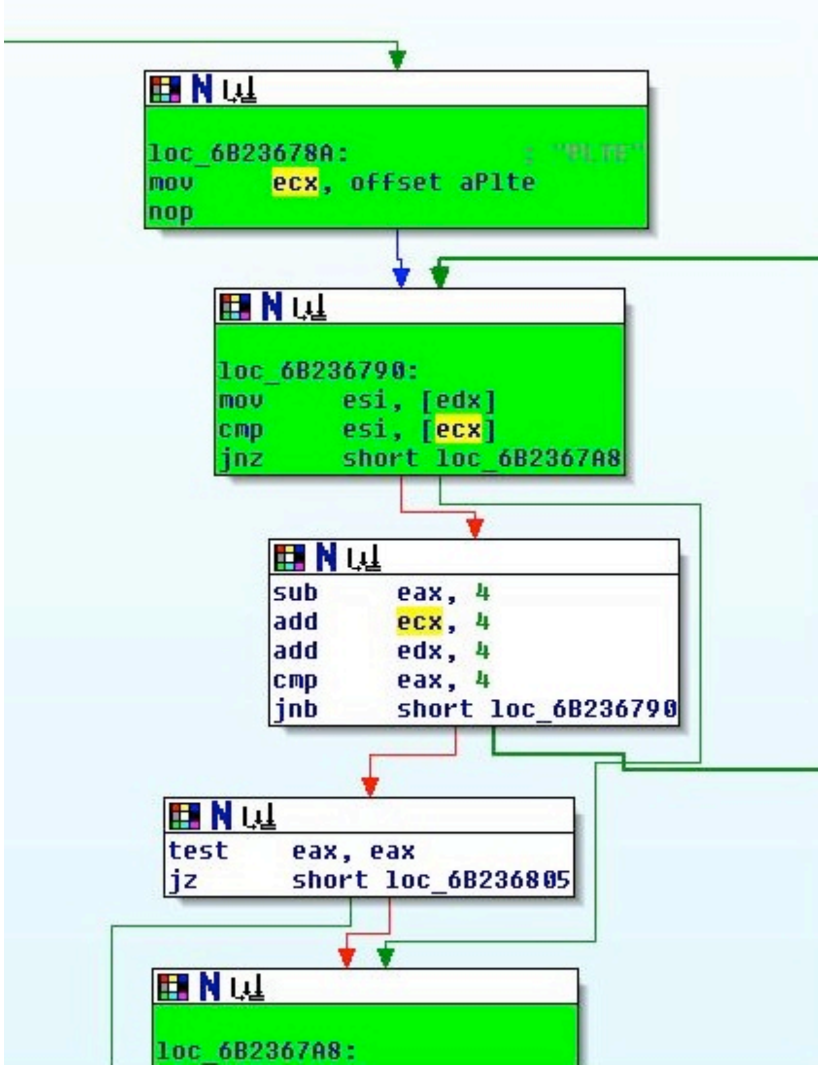
- Pai Mei is only as good as what IDA give it
  - If IDA misidentifies data as code, bad things happen!
- Some anti-debugging measures screw it up
- Have to know the DLL you're interested in
  - Or load them all
- For large binaries, it can be slow to set all the breakpoints
  - For this library, it takes a few minutes

Increasing Code Coverage

- Lack of code coverage is a bad thing
  - Can't find bugs in code you're not executing
- How do you increase code coverage?
- Basically three ways
  - Manually
  - Dynamically using run time information
  - Automatically from static information

Manually

- You can imagine looking at the PNG code and figuring out how to get more code coverage.



Another Example

- Freeciv 2.0.9, a free multiplayer game similar to Civilization

- Don't ever play this on a computer you care about

Steps to Code Coverage

- Get the Windows binary - no cheating
- Disassemble it
- Dump the PIDA file
- Launch civserver.exe
- Attach with Pai Mei's Pstalker
- Capture a netcat connection to it
- Filter this out (551 of 36,183 BB's - 2%)
- Trace the fuzzing!

GPF

- Great, general purpose mutation-based fuzzer
- Works on packet captures
- Replays packets while injecting faults
- User can manually tell GPF about the structure of the data in the packets
    - Aids in the anomaly injection
- Many modes of operation

Fuzz FreeCiv

- Start up the game, play a bit
- Capture the packets to a file
- Convert the PCAP file to a GPF file

```
./GPF -C freeciv_reg_game.pcap freeciv_reg_game.gpf
```

- Fire up GPF (main mode)
  - Main mode replaces some packets with random data

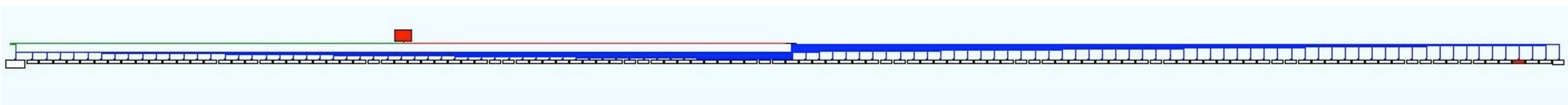
```
./GPF -G l ../freeciv_reg_game.gpf client <IP ADDRESS> 5555 ? TCP
kj3874jff 1000 0 + 0 + 00 01 09 43 close 0 1 auto none
```

FreeCiv Sucks

- Not designed to be fuzzed :)
- Need to add a sleep to GPF so FreeCiv can keep up
- Fuzz overnight...
- I recorded 96 functions during fuzzing
  - 614 / 36183 basic blocks
- Import data back to IDA
- Look for places to increase code coverage

I See One!

- A big switch statement I only hit once



- Tracing back reveals this switch is controlled by the third byte of the packet

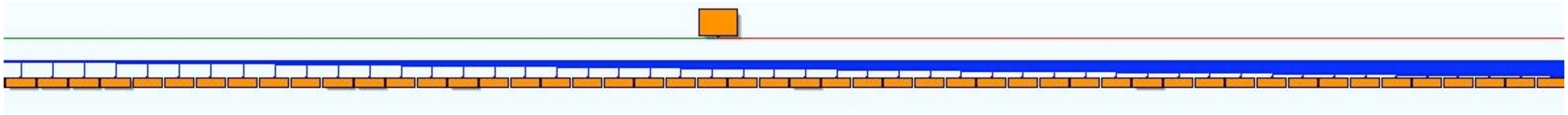
Back to GPF

- Up until now we've basically been sending random data
- Using Pai Mei, we observe that the third byte is important
- We modify GPF to make sure it changes the third byte
- We've added a little structure to our random data

```
./bin/GPF -G l freeciv_reg_game.gpf client <IP ADDRESS> 5555 ?
TCP kj3874jff 1000 0 + 2 2 00 01 255 41 finish 0 1 auto none
```

Better Code Coverage

- 2383 basic blocks covered (after filtering)
  - Compare this to 614 with the first fuzz run
  - 4x improvement



- All cases taken in switch (as expected)
- However, still no bugs...

Manual Method Explained

- Send mostly random data
- Examine code coverage to see what structure in the data is important
- Send data which has some elements set but some mostly random parts
- Rinse and Repeat

Fuzzing Beyond the 3rd Byte

- This command replaces the bytes 3 through 10 of each packet, one at a time, with all possible values from 0 to 255

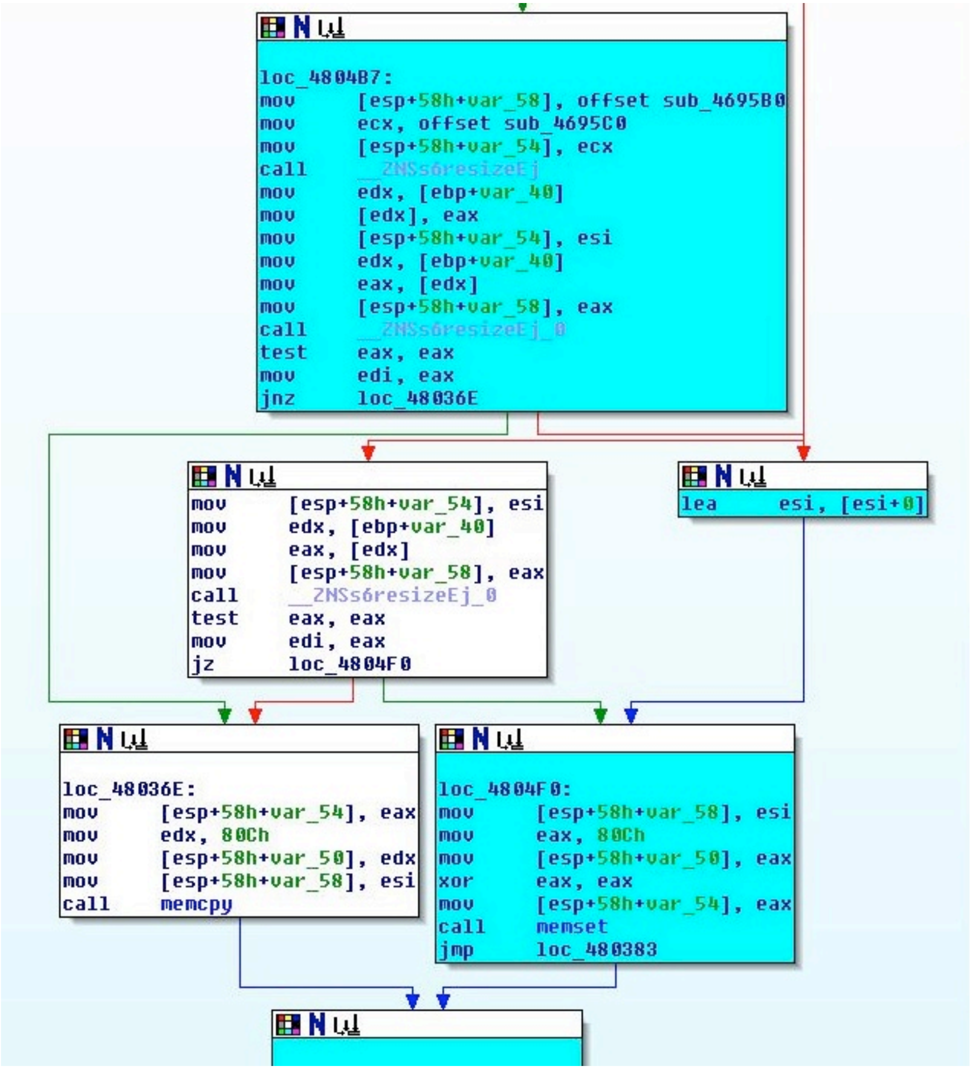
```
./GPF -G l ../freeciv_reg_game.gpf client <IP ADDRESS> 5555 ?
TCP kj3874jff 1000 0 + 2 10 00 01 255 41 finish 0 1 auto none
```

- This will ensure that all the cases in the switch statement are hit and each case will have some random data
- After a bit, CPU is pegged: Memory consumption bug!

Dig Deeper

- Following the methodology, fix the 3rd byte to, say 0x47
- Send in random data to that part of the program
- See what you missed
- Try to do better

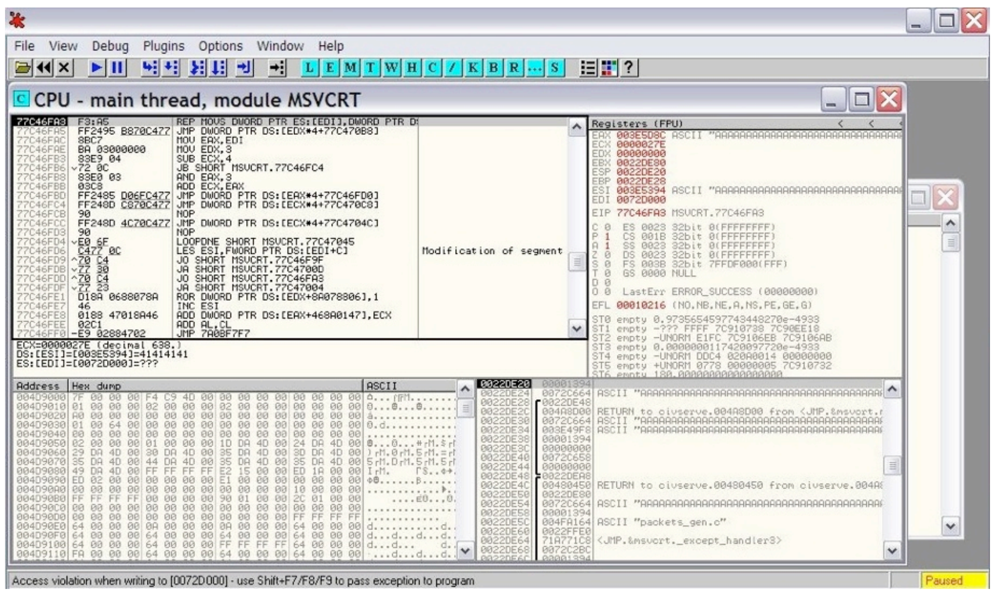
Missed Some Spots



Heap Overflow

- Can get a heap overflow if you send the following packet:

27 2e | 2f | 0c | 00 00 13 94 | 41 41 41 41 41...
Length of Packet Length of memcpy Data



Bugs In FreeCiv Aren't a Huge Deal

- Fun for hacking your friends
- Also MetaServer is nice

Freeciv servers around the world

Host	Port	Version	Patches	State	Players	Topic	Last Update	Players Available
88-134-81-104-dynip.superkabel.de	5555	2.0.9	none	Running	32		2m	30
p5B20C598.dip.t-dialin.net	5560	2.0.9	Warserver - PepServer 0.9.5 devel	Pregame	0	NEW GAME	27s	0
pool-72-90-153-69.nwrknj.east.verizon.net	5555	2.0.9	none	Running	10	New Game	2m	8
pool-72-90-153-69.nwrknj.east.verizon.net	5556	2.0.9	none	Running	2	New Game	2m	2
pool-72-90-153-69.nwrknj.east.verizon.net	5557	2.0.9	none	Running	9	New Game	16s	9
wsip-70-182-164-206.ks.ks.cox.net	5555	2.0.9	none	Pregame	0		39s	0
wsip-70-184-212-144.om.om.cox.net	5555	2.0.9	none	Running	5		2m	5
ww10.ultico.de	5551	2.0.9	Warserver - PepServer 0.9.5 devel	Pregame	0	NEW GAME	2s	0
ww10.ultico.de	5552	2.0.9	Warserver - PepServer 0.9.5 devel	Game Ended	5	NEW GAME	16s	5
ww10.ultico.de	5553	2.0.9	Warserver - PepServer 0.9.5 devel	Pregame	0	NEW GAME	48s	0
ww10.ultico.de	5554	2.0.9	Warserver - PepServer 0.9.5 devel	Pregame	0	NEW GAME	49s	0
ww10.ultico.de	5555	2.0.9	Warserver - PepServer 0.9.5 devel	Pregame	0	NEW GAME	20s	0

Dynamically Generating Better Test Cases

- Manually improving code coverage is, uh, “time intensive”
- Need to automate the process
- Autodafe kinda does this
- But I prefer another of Jared Demott’s tools....

EFS

- Uses Pai Mei Pstalker to record code coverage
- Uses Genetic Algorithms to generate new test cases based on code coverage feedback

Genetic Algorithms

- Technique to find approximate solution to optimization problems
- Inspired by evolutionary biology
  - Define fitness of an organism (test case)
  - Must define how to recombine two organisms
  - Must define how to mutate a single organism
- Lots more complexity but that is the basics

GA example

- `f(x) = -x * (x - 10000)`
- Use “single point crossover” of binary representation of numbers for recombination

```
677  : 00000000000000000000001010100101
9931 : 00000000000000000010011011001011
-------------------------------------------------------------
651  : 00000000000000000000001010001011
```

- Flip a bit 10% of the time for mutation
- Fitness is the value in the function

In Practice

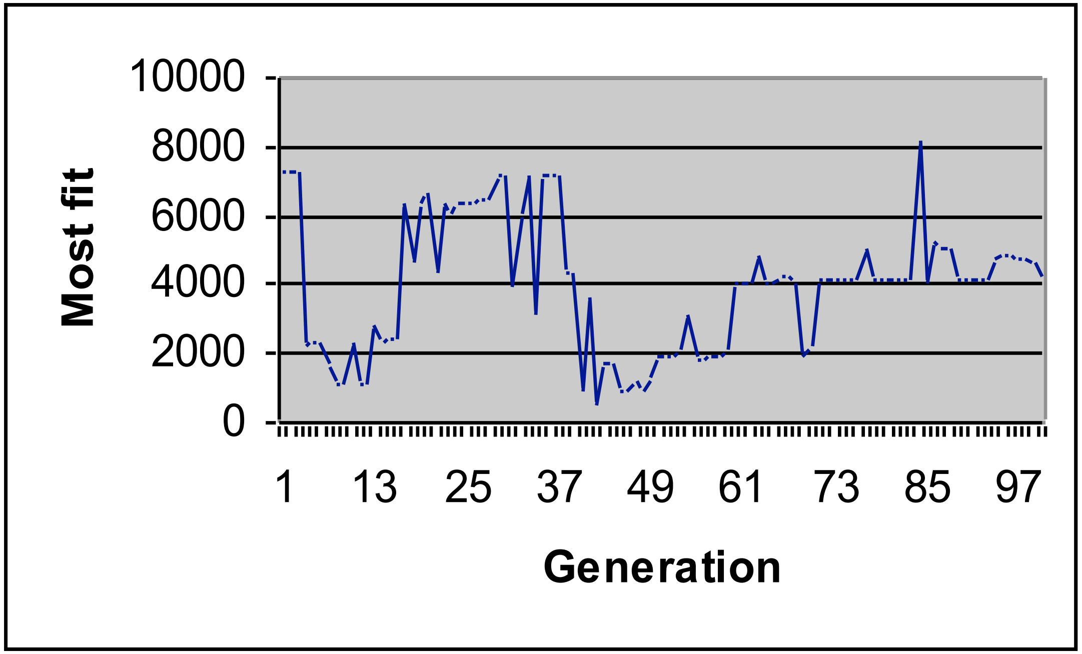
- Running it for a few generations gives

134 (1322044)	651 (6086199)	485 (4614775)	7653 (17961591)	1354 (11706684)	7654 (17956284)	134 (1322044)	7302 (19700796)
1354 (11706684)	7652 (17966896)	134 (1322044)	7653 (17961591)	7302 (19700796)	390 (3747900)	1350 (11677500)	134 (1322044)
390 (3747900)	7302 (19700796)	134 (1322044)	134 (1322044)	70 (695100)	134 (1322044)	1350 (11677500)	134 (1322044)
134 (1322044)	134 (1322044)	268 (2608176)	134 (1322044)	1350 (11677500)	134 (1322044)	134 (1322044)	2182 (17058876)
134 (1322044)	134 (1322044)	134 (1322044)	2182 (17058876)	1618 (13562076)	134 (1322044)	134 (1322044)	2316 (17796144)
134 (1322044)	1618 (13562076)	1612 (13521456)	132 (1302576)	2316 (17796144)	134 (1322044)	2322 (17828316)	1158 (10239036)

- The optimum value is 5000

GA Approaches the Solution

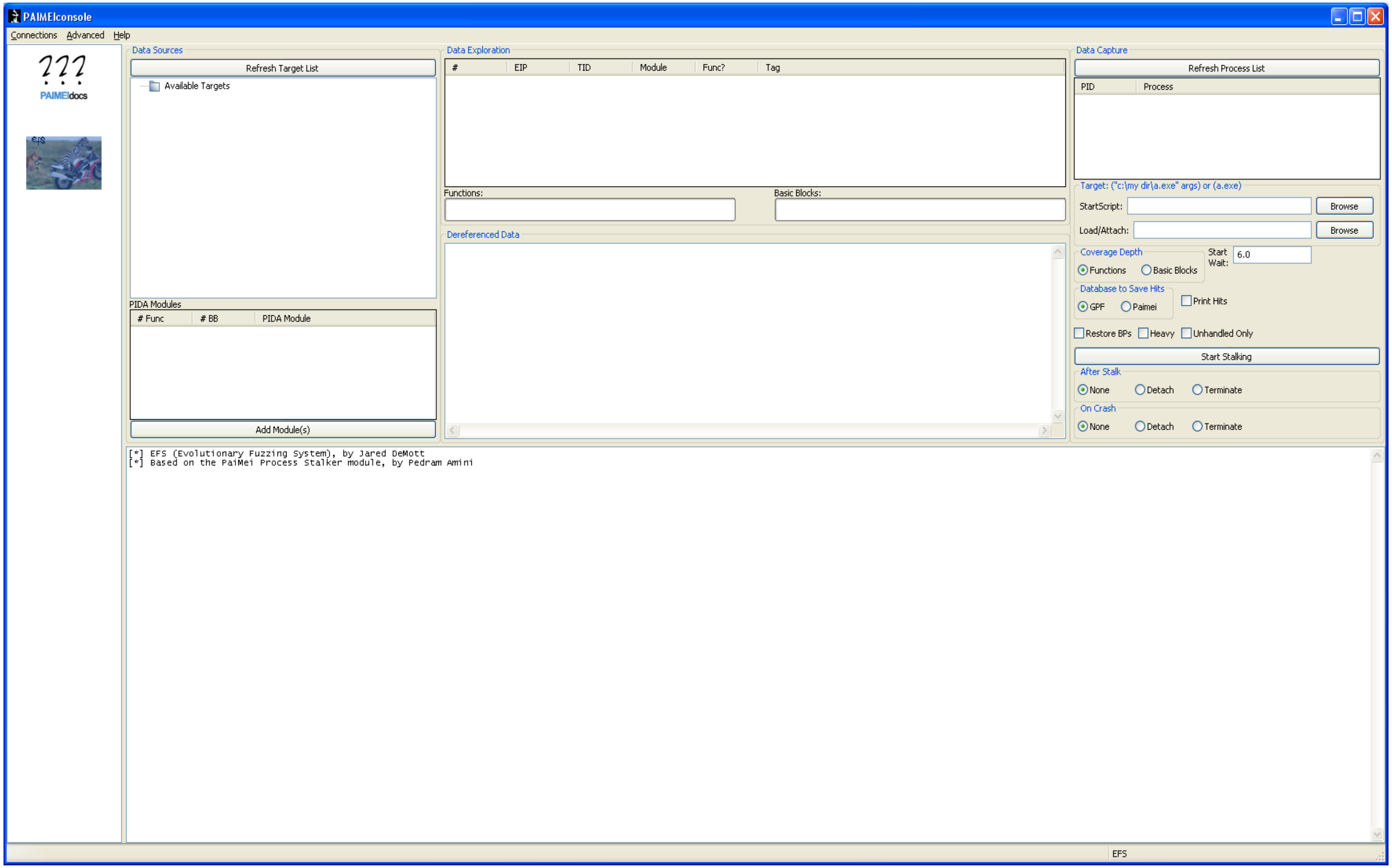
- Generation vs most fit individual
- Approaches the solution



EFS and GA's

- Fitness function: How many functions were covered by the test case (in reality a more elaborate measure is used)
- For breeding, tends to choose the most fit individuals
- Recombination: single point crossover that respects "protocol tokens"
- Mutation: portions of data replaced with fuzzing heuristics

Obligatory Screenshot



Running EFS

- Still needs a PIDA file
- Connect to database
- Add PIDA file to module list
- Enter pathname to application in Load/Attach window
- Choose Connections->Fuzzer Connect
  - Hit “Listen”
- On Client

```
./GPF -E <IP ADDRESS> root <PASSWORD> 0 0 <IP ADDRESS> 31338 funcs client <IP ADDRESS>
5555 ? TCP 800000 20 low AUTO 4 25 Fixed 10 Fixed 10 Fixed 35 3 5 9 none none no
```

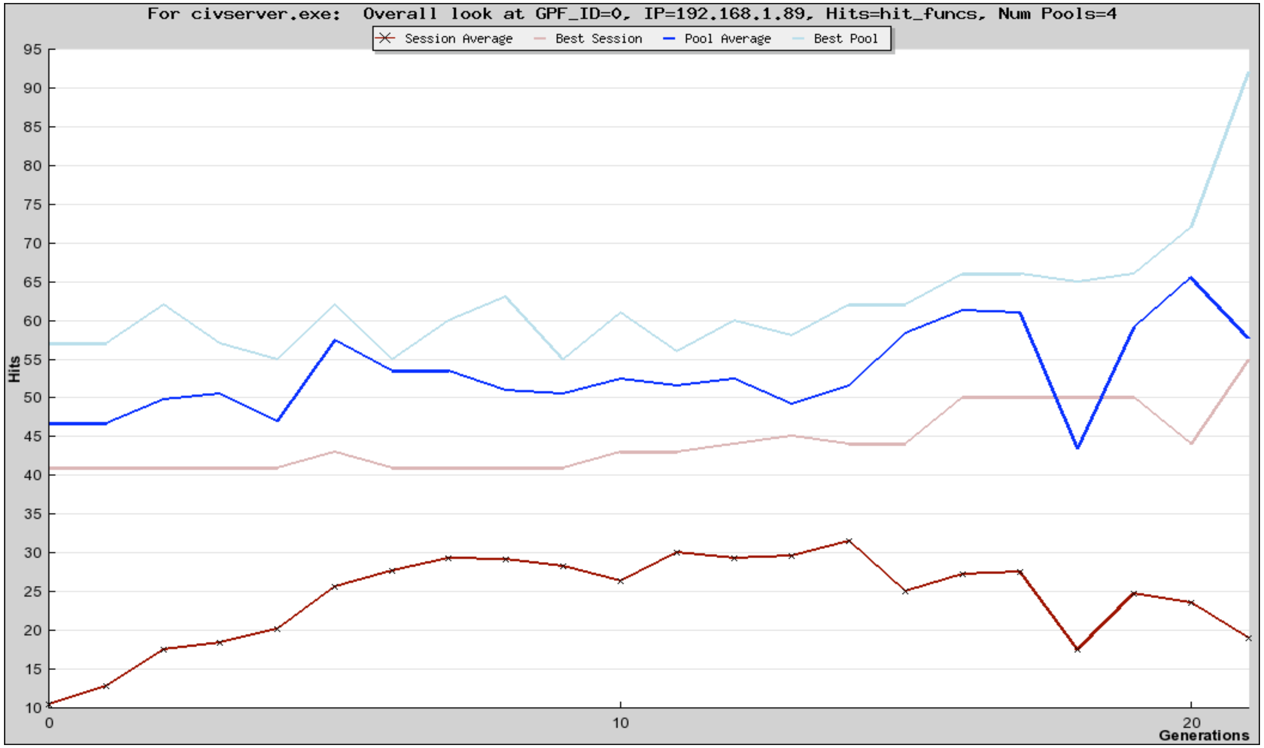
What You See

```
Successfully played generation 0. Saving to mysqldb.
Processing Generation 0 ...
Done processing. Time to play and process: 100 total
evaluations in
1001 seconds.
10.01 sec/eval
That's 16.683 mins or 0.278 hrs.

Successfully played generation 1. Saving to mysqldb.
Processing Generation 1 ...
Done processing. Time to play and process: 200 total
evaluations in
1887 seconds.
9.44 sec/eval
That's 31.450 mins or 0.524 hrs.
```

Does It Work?

- The light blue line indicates the most fit pool of testcases
- Code coverage is (slowly) improving



Caveats

- Still experimental
- GA's can get stuck in "local maxima"
- GA's have so many parameters (population size, initial population, mutation percentage, etc), hard to optimize

Statically Generating Code Coverage

- GA's attempt to provide an approximating solution to a difficult problem
- We have the binary, we have the control flow graph, we have the disassembly...
- What if we "solve" the problem exactly?

Existing Work

- Microsoft Research has a tool that generates code coverage maximizing test cases from binaries
  - ftp://ftp.research.microsoft.com/pub/tr/TR-2007-58.pdf
- Catchcov (built on Valgrind) does something similar to try to find integer overflows
- Greg Hoglund has something which tries to do this
- Nothing freely available

General Idea

- Identify where user supplied data enters the program
- Data needs to be traced (symbolically) and branch point's dependence on initial data recorded
- These equations need to be solved, i.e. inputs need to be generated which can go down either branch at each branch point.

Example

- Input comes in through `argv[1]`
- `test()` takes an this value as an int
- 3 possible paths through the program

```
int test(int x){
        if(x < 10){
                if(x > 0){
                        return 1;
                }
        }
        return 0;
}

int main(int argc, char *argv[]){
        int x = atoi(argv[1]);
        return test(x);
}
```

Tracing the Data

- Use Valgrind or PyEmu?
- In this trivial example, we'll just do it by hand.
- The constraints would look something like

```
x >= 10
0 < x < 10
x <= 0
```

- In real life, there would be thousands of such constraints

Solve the Constraints

- Can use a Boolean satisfiability solver (SAT)
- One such solver is STP
  - Constraints expressed as bit vector variables
  - Bitwise operators like AND, OR, XOR
  - Arithmetic functions like +, =, *
  - Predicates like =, <, >

In the STP Language

```
x : BITVECTOR(32);
QUERY(BVLT(x,0hex0000000a));
```

```
x : BITVECTOR(32);
ASSERT(BVLT(x,0hex0000000a));
QUERY(BVGT(x,0hex00000000));
```

```
x : BITVECTOR(32);
ASSERT(BVLT(x,0hex0000000a));
QUERY(BVLE(x,0hex00000000));
```

Solving These Gives

- This gives the test cases `x={12, 0, 4}`
- These give maximal code coverage

```
$ ./stp -p q1
Invalid.
ASSERT( x  = 0hex0000000C  );
$ ./stp -p q2
Invalid.
ASSERT( x  = 0hex00000000  );
$ ./stp -p q3
Invalid.
ASSERT( x  = 0hex00000004  );
```

Using This Technique

- Very sophisticated constraints, such as those that found the Freeciv bug, could be solved (sometimes)
- Optimum test cases can be generated without executing the application
- Combining dynamic and static approaches can optimize fuzzing

Conclusion

- Fuzzing is easy, until you really try it
- Code coverage is a tool that can be used to try to measure and improve fuzzing
- You won’t find any bugs in code you haven’t tested
- Increasing code coverage can be difficult and time consuming but new tools are coming to make this easier

References

- http://en.wikipedia.org/wiki/Fuzz_testing
- Make My Day: Just Run A Web Scanner, Toshinari Kureha, BH-EU-07
- How Smart is Intelligent Fuzzing - or - How Stupid is Dumb Fuzzing, Charlie MIller, DEFCON 2007
- Robustness Testing Code Coverage Analysis, Teno Rontti, Masters Thesis
- How to Misuse Code Coverage, Brian Marick, http://www.testing.com/writings/coverage.pdf
- ProxyFuzz: http://theartoffuzzing.com/joomla/index.php?option=com_content&task=view&id=21&Itemid=40
- STP: http://theory.stanford.edu/~vganesh/stp.html
- SPIKE: http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- lcov: http://ltp.sourceforge.net/coverage/lcov.php
- GPF and EFS: http://www.vdalabs.com/tools/efs_gpf.html

Questions?

- Please contact me at: cmiller@securityevaluators.com