PA193 - Secure coding principles and practices

Dynamic analysis, fuzzing

Petr Švenda

svenda@fi.muni.cz @rngsec

CRoCS

Centre for Research on
Cryptography and Security

Overview

- Lecture:
  - Dynamic analysis of programs for potential bugs
  - Memory analysis
  - Fuzzing (blackbox testing)
  - Tools

DYNAMIC ANALYSIS

What can dynamic analysis provide

- Dynamic analysis compile and execute tested program
  - real or virtualized processor
- Inputs are supplied and outputs are observed
  - sufficient number of inputs needs to be supplied
  - code coverage should be high
- Memory, function calls and executed operations can be monitored and evaluated
  - invalid access to memory (buffer overflow)
  - memory leak or double free (memory corruption)
  - calls to potentially sensitive functions (violation of policy)

Techniques used by dynamic analysis

- Debugger (full control over memory read/write, even ops)
- Insert data into program input points (integration tests, fuzzing…)
  - stdin, network, files…
- Insert manipulation proxy between program and library (dll stub, memory)
- Trace of program’s external behavior (linux strace)
- Change source code (instrumentation, logging…)
- Change of application binary
- Run in lightweight virtual machine (Valgrind)
- Run in full virtual machine
- Follow propagation of specified values (Taint analysis)
- Mocking (create additional input points into program)
- Restrict programs environment (low memory, limited file descriptors, limited rights…)

DEBUGGING SYMBOLS

Release vs. Debug

- Optimizations applied (compiler-specific settings)
  - gcc –Ox (http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)
    - -O0 no optimization (Debug)
    - -O1 –g / -Og debug-friendly optimization
    - -O3 heavy optimization
  - msvc /Ox /Oi (http://msdn.microsoft.com/en-us/library/k1ack8f1.aspx)
    - MSVS2010: Project properties→C/C++→optimizations
- Availability of debug information (symbols)
  - gcc –g
    - symbols inside binary
  - msvc /Z7, /Zi
    - symbols in detached file ($projectname.pdb)

Stripping out debug symbols

- Debug symbols are of great help for an “attacker”
  - key called NSAKey in ADVAPI.dll? (Crypto 1998)
  - http://www.heise.de/tp/artikel/5/5263/1.html
- Always strip out debug symbols in released binary
  - check compiler flag
  - Linux: run **`file`** or **`objdump --syms`** command (stripped/not stripped)
  - Windows: DependencyWalker

VALGRIND SUITE

Valgrind http://www.valgrind.org/

- Suite of multiple tools (`valgrind --tool=<toolname>`)
- Memcheck - memory management dynamic analysis
  - most commonly used tool (memory leaks)
  - replaces standard C memory allocator with its own implementation and check for memory leaks, corruption (additional guards blocks)...
  - dangling pointers, unclosed file descriptors, uninitialized variables
  - http://www.valgrind.org/docs/manual/mc-manual.html
- Massif – heap profiler
- Hellgrind - detection of concurrent issues
- Callgrind – generation of all graphs
- ...

Valgrind – core options

- Compile with debug symbols
  - `gcc -std=c99 -Wall -g -o program program.c`
  - will allow for more context information in Valgrind report
- Run program with Valgrind attached
  - `valgrind <options> ./program`
  - program cmd line arguments (if any) can be passed
  - `valgrind -v --leak-check=full ./program arg1`
- Trace also into sub-processed
  - `--trace-children=yes`
  - necessary for multi-process / threaded programs
- Display unclosed file descriptors
  - `--track-fds=yes`

Memcheck – memory leaks

- Detailed report of memory leaks checks
  - --leak-check=full
- Memory leaks
  - *Definitely lost*: memory is directly lost (no pointer exists)
  - *Indirectly lost*: only pointers in lost memory points to it
  - *Possibly lost:* address of memory exists somewhere, but might be just randomly correct value (usually real leak)

Memcheck – uninitialized values

- Detect usage of uninitialized variables
  - `-undef-value-errors=yes` (default)
- Track from where initialized variable comes from
  - `--track-origins=yes`
  - introduces high performance overhead

Memcheck – invalid reads/writes

- Writes outside allocated memory (buffer overflow)
- Only for memory located on heap!
  - allocated via dynamic allocation (malloc, new)
- Will NOT detect problems on stack or static (global) variables
  - https://en.wikipedia.org/wiki/Valgrind#Limitations_of_Memcheck
- Writes into already de-allocated memory
  - Valgrind tries to defer reallocation of freed memory as long as possible to detect subsequent reads/writes here

EXAMPLES OF ANALYSIS

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;
  Stack[100] = 0;

  for (int i = 0; i <= 5; i++) Stack [i] = 0;

  int* array = new int[5];
  array[100] = 0;

  arrayStack[100] = 0;
  arrayHeap[100] = 0;

  for (unsigned int i = 0; i <= arrayStackLen; i++) {
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) {
      arrayHeap[i] = 0;
  }

  return 0;
}
```

```
int main(void) {
  int arrayStack[5];
  int* arrayHeap = new int[5];
  memcheckFailDemo(arrayStack, 5, arrayHeap, 5);
  return 0;
```

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;  /* Error - Static[100] is out of bounds */
  Stack[100] = 0;  /* Error - Stack[100] is out of bounds */

  for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

  int* array = new int[5];
  array[100] = 0; /* Error - array[100] is out of bounds */

  arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
  arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

  for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one *
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
      arrayHeap[i] = 0;
  }
  /* Problem Memory leak –
  return 0;
}
```

```
int main(void) {
  int arrayStack[5];
  int* arrayHeap = new int[5];
  memcheckFailDemo(arrayStack, 5, arrayHeap,
  return 0;
}
```

Problems detected – compile time

- g++ -ansi -Wall -Wextra -g -o test test.cpp
  - clean compilation

- MSVC (Visual Studio 2012) /W4
  - only one problem detected, `Stack[100] = 0;`

```
test.cpp(56): error C4789: buffer 'Stack' of size 20 bytes will
be overrun; 4 bytes will be written starting at offset 400
```

- MSVC (Visual Studio 2015) /W4
  - No problem reported (detection moved into PREFast)

MSVC /W4

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack,
        int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;  /* Error - Static[100] is out of bounds */
  Stack[100] = 0;  /* Error - Stack[100] is out of bounds */

  for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

  int* array = new int[5];
  array[100] = 0; /* Error - array[100] is out of bounds */

  arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
  arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

  for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one *
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
      arrayHeap[i] = 0;
  }
  /* Problem Memory leak - array */
  return 0;
}
```

Visual Studio 2015 & PREfast & SAL

```
int memcheckFailDemo(
  _Out_writes_bytes_all_(arrayStackLen) int* arrayStack,
  unsigned int arrayStackLen,
  _Out_writes_bytes_all_(arrayHeapLen) int* arrayHeap,
  unsigned int arrayHeapLen);
```

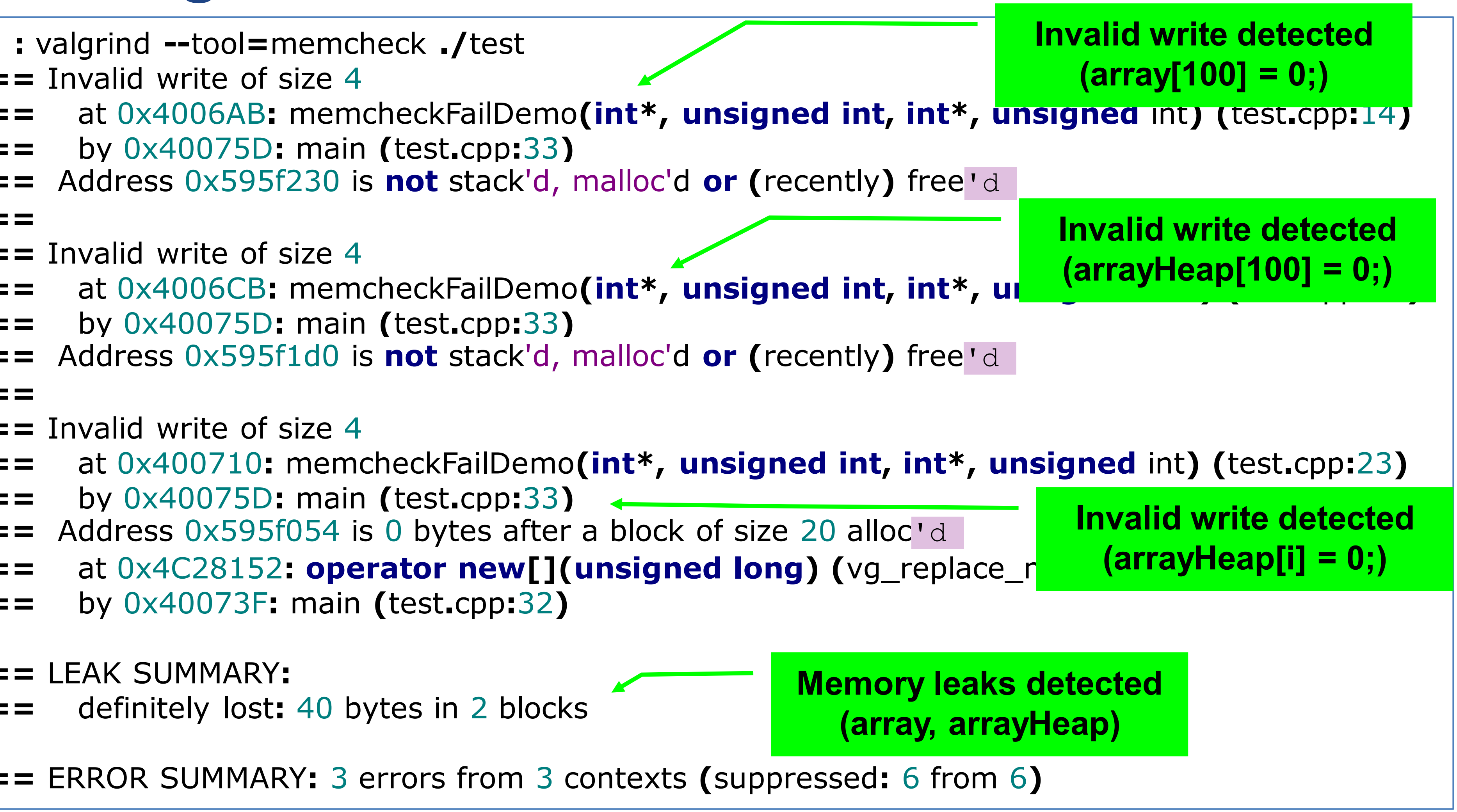
```
test.cpp(11): warning : C6200: Index '100' is out of valid index
   range '0' to '4' for non-stack buffer 'int * Static'.
test.cpp(14): warning : C6201: Index '5' is out of valid index
   range '0' to '4' for possibly stack allocated buffer 'Stack'.
test.cpp(11): warning : C6386: Buffer overrun while writing to 'Static':
   the writable size is '20' bytes, but '404' bytes might be written.
test.cpp(17): warning : C6386: Buffer overrun while writing to 'array':
   the writable size is '5*4' bytes, but '404' bytes might be written.
test.cpp(23): warning : C6386: Buffer overrun while writing to 'arrayStack':
   the writable size is '_Old_2`arrayStackLen' bytes, but '8' bytes might be written.
test.cpp(26): warning : C6386: Buffer overrun while writing to 'arrayHeap':
   the writable size is '_Old_2`arrayHeapLen' bytes, but '8' bytes might be written.
```

Visual Studio 2015 & PREfast & SAL

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;  /* Error - Static[100] is out of bounds */
  Stack[100] = 0;  /* Error - Stack[100] is out of bounds */

  for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

  int* array = new int[5];
  array[100] = 0; /* Error - array[100] is out of bounds */

  arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
  arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

  for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one *
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
      arrayHeap[i] = 0;
  }
  /* Probl
  return
}
```

```
/* Error - still off by one, but not detected by SAL */
for (unsigned int i = 0; i < arrayStackLen + 1; i++) {
  arrayStack[i] = 0;
}
```

Valgrind --tool=memcheck



```
 : valgrind --tool=memcheck ./test
== Invalid write of size 4
==    at 0x4006AB: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:14)
==    by 0x40075D: main (test.cpp:33)
==  Address 0x595f230 is not stack'd, malloc'd or (recently) free'd
==
== Invalid write of size 4
==    at 0x4006CB: memcheckFailDemo(int*, unsigned int, int*, ur
==    by 0x40075D: main (test.cpp:33)
==  Address 0x595f1d0 is not stack'd, malloc'd or (recently) free'd
==
== Invalid write of size 4
==    at 0x400710: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:23)
==    by 0x40075D: main (test.cpp:33)
==  Address 0x595f054 is 0 bytes after a block of size 20 alloc'd
==    at 0x4C28152: operator new[](unsigned long) (vg_replace_r
==    by 0x40073F: main (test.cpp:32)

== LEAK SUMMARY:
==    definitely lost: 40 bytes in 2 blocks

== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)
```

Invalid write detected (array[100] = 0;)

Invalid write detected (arrayHeap[100] = 0;)

Invalid write detected (arrayHeap[i] = 0;)

Memory leaks detected (array, arrayHeap)

Valgrind --tool=memcheck

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;  /* Error - Static[100] is out of bounds */
  Stack[100] = 0;  /* Error - Stack[100] is out of bounds */

  for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

  int* array = new int[5];
  array[100] = 0; /* Error - array[100] is out of bounds */

  arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
  arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

  for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one *
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
      arrayHeap[i] = 0;
  }
  /* Problem Memory leak - array */
  return 0;
}
```

Valgrind --tool=exp-sgcheck

Invalid write detected

```
for (int i = 0; i <= 5; i++) Stack[i] = 0;
```

```
==15979== Invalid write of s
==15979==    at 0x40067C: memcheckFailDemo(int*, unsigned int, int*,
  unsigned int) (test.cpp:11)
==15979==    by 0x40075D: main (test.cpp:33)
==15979==  Address 0x7fefffe34 expected vs actual:
==15979==  Expected: stack array "Stack" of size 20 in this frame
==15979==  Actual:   unknown
==15979==  Actual:   is 0 afte
==15979==
==15979== Invalid write of size 4
==15979==    at 0x4006E5: memcheckFailDemo(int*, unsigned int, int*,
  unsigned int) (test.cpp:20)
==15979==    by 0x40075D: main (test.cpp:33)
==15979==  Address 0x7fefffe74 expected vs actual:
==15979==  Expected: stack array "arrayStack" of size 20 in frame 1 back from here
==15979==  Actual:   unknown
==15979==  Actual:   is 0 after Expected
==15979==
==15979==
==15979== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 28 from 28)
```

Invalid write detected

```
... arrayStack[i] = 0;
```

Valgrind --tool=exp-sgcheck

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  Static[100] = 0;  /* Error - Static[100] is out of bounds */
  Stack[100] = 0;  /* Error - Stack[100] is out of bounds */

  for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

  int* array = new int[5];
  array[100] = 0; /* Error - array[100] is out of bounds */

  arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
  arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

  for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one *
      arrayStack[i] = 0;
  }
  for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
      arrayHeap[i] = 0;
  }
  /* Problem Memory leak – array */
  return 0;
}
```

(MSVS 2012) _CrtDumpMemoryLeaks();

```
Detected memory leaks!
Dumping objects ->
{155} normal block at 0x00600AD0, 20 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
{154} normal block at 0x00600A80, 20 bytes long.
 Data: <                > 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Object dump complete.
```

Tools - summary

- *Compilers* (MSVC, GCC) will miss many problems
- *Compiler flags (/RTC* and */GS;* **`-fstack-protector-all`***)* flags
  - detect (some) stack based corruptions at runtime
  - additional preventive flags /DYNAMICBASE (ASLR) and /NXCOMPAT (DEP)
- *Valgrind memcheck*
  - will not find stack based problems, only heap corruptions (dynamic allocation)
- *Valgrind exp-sgcheck*
  - will detect stack based problem, but miss first (possibly incorrect) access
- *Cppcheck*
  - detect multiple problems (even memory leaks), but mostly limited to single function
- *PREfast* will find some stack based problems, limited to single function
- *PREfast with SAL* annotations will find additional stack and some heap problems, but not all

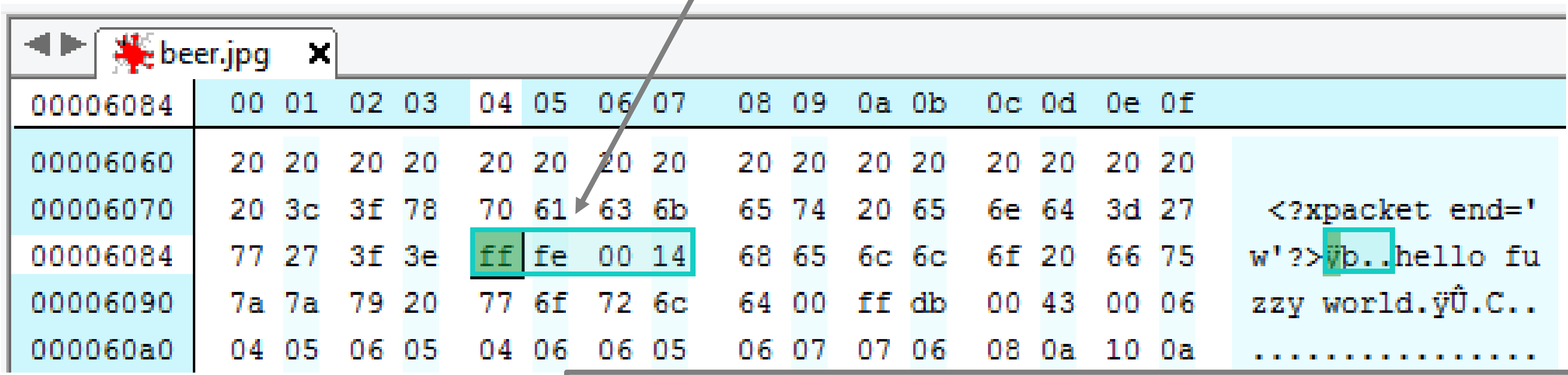
FUZZING (BLACKBOX)

Beer
HELPING ENGLISH PEOPLE
HAVE SEX SINCE 1842!
THE ENGLISH PUB
Austurvöllur

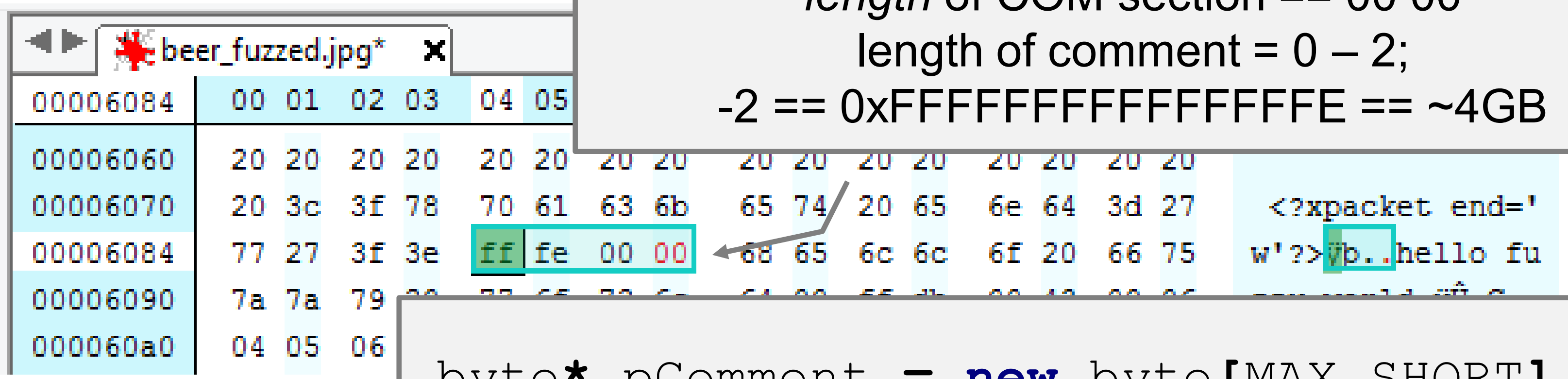
MS04-028: Microsoft's JPEG GDI+ vulnerability (2004)

What is wrong?

Tag ‘ff fe’ + *length* of COM section
length of comment = *length* – 2;
strlen(“hello fuzzy world”) == ?



```
00006084  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f

00006060  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20
00006070  20 3c 3f 78  70 61 63 6b  65 74 20 65  6e 64 3d 27   <?xpacket end='
00006084  77 27 3f 3e  ff fe 00 14  68 65 6c 6c  6f 20 66 75  w'?>ÿþ..hello fu
00006090  7a 7a 79 20  77 6f 72 6c  64 00 ff db  00 43 00 06  zzy world.ÿÛ.C..
000060a0  04 05 06 05  04 06 06 05  06 07 07 06  08 0a 10 0a  ................
```



```
00006084  00 01 02 03  04 05

00006060  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20
00006070  20 3c 3f 78  70 61 63 6b  65 74 20 65  6e 64 3d 27   <?xpacket end='
00006084  77 27 3f 3e  ff fe 00 00  68 65 6c 6c  6f 20 66 75  w'?>ÿþ..hello fu
00006090  7a 7a 79
000060a0  04 05 06
```

length of COM section == 00 00
length of comment = 0 – 2;
-2 == 0xFFFFFFFFFFFFFFFE == ~4GB

```
byte* pComment = new byte[MAX_SHORT];
memcpy(pComment, buffer, length);
```

I love GDI+ vulnerability because…

- Lack of proper ir
- Type signed/uns
- Type overflow
- Buffer overflow
- Heap overflow
- Source code wa
- Huge impact (cc
- Easily exploitabl

FOUND BY FUZZING ☺

INTRO TO FUZZING

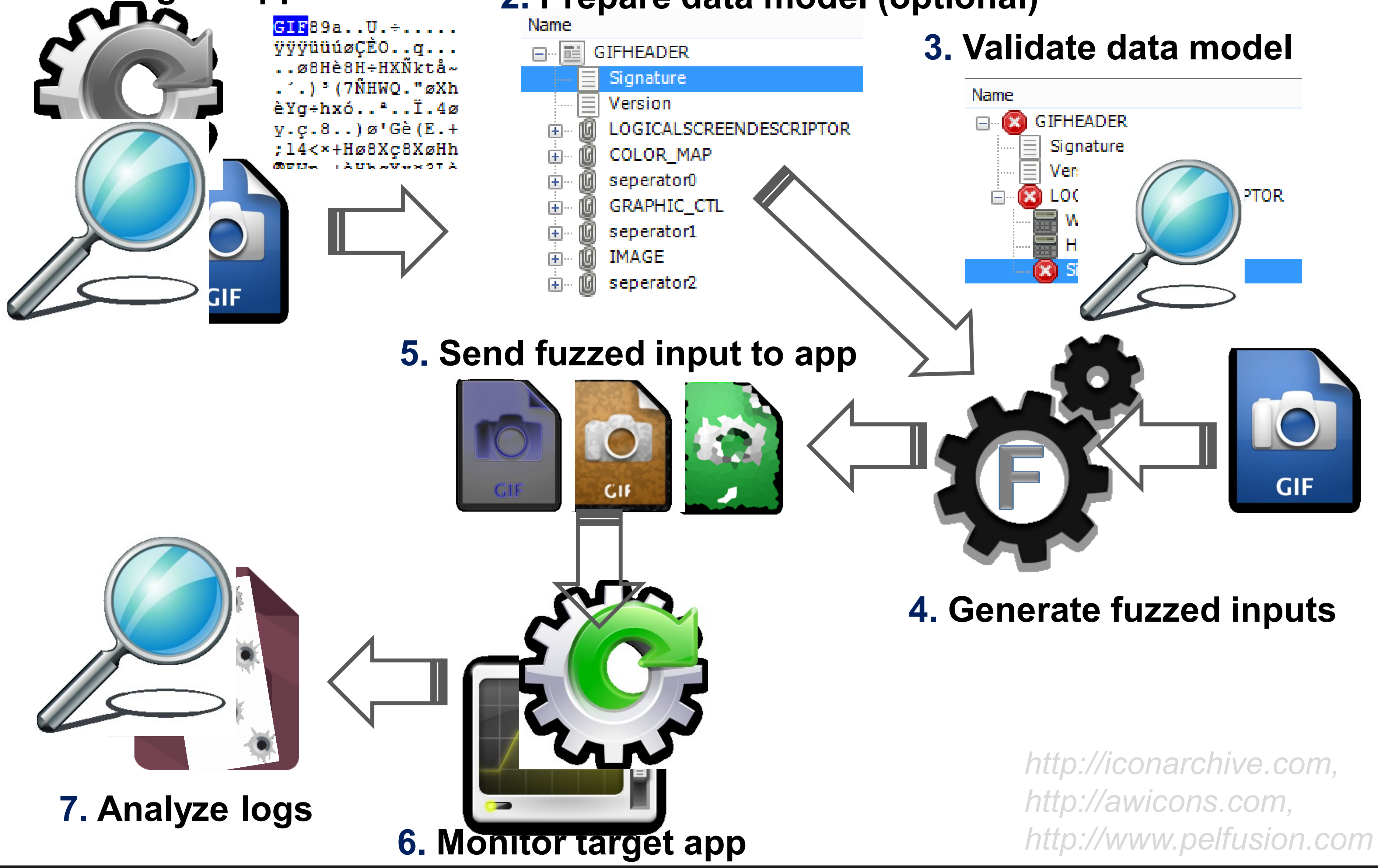
Very simple fuzzer

```
cat /dev/random | ./target_app
```

What do you miss here?

What is missing?

- Where fuzzing fits in development process? (developer side, CI, SDL)
- What type of bugs fuzzing tends to find?
- What apps can be fuzzed?
- How to detect that app mishandled fuzzed input ("hit")? (crash, signal, exception, error…)
- How to react on detected "hit"? (save seed and crashing inputs, bucketing of inputs)
- How to create more meaningful inputs then random bytes? (valid inputs, proxy)
- How to fuzz non-binary inputs? (string patterns, regexpr, mouse movements…)
- How to fuzz applications without input as files? (http requests, dll injection, ZAP example)
- How to fuzz efficiently? (known problematic values (fuzz vectors))
- How to fuzz files/inputs with defined structure? (grammar, example Peach)
- How to make fuzzer protocol-aware? (Peach example)
- How to fuzz state-full protocols? (proxy like fuzzing)
- How to analyse and react on detected hits?
- Which tools one can use?
- How to detect less visible "hits"? (side-channels)
- What else can we fuzz? (test coverage testing, DDOS resiliency, hardware inputs)



1. Investigate app in/out

```
GIF89a..U.÷.....
ÿÿÿüüú¢ÇÈO..q...
..ø8Hè8H÷HXÑktå~
.´.)³(7ÑHWQ."øXh
èYg÷hxó..ª..Ï.4ø
y.ç.8..)ø'Gè(E.+
;14<×+Hø8Xç8XøHh
```

2. Prepare data model (optional)

```
Name
⊟ GIFHEADER
    Signature
    Version
⊞ LOGICALSCREENDESCRIPTOR
⊞ COLOR_MAP
⊞ seperator0
⊞ GRAPHIC_CTL
⊞ seperator1
⊞ IMAGE
⊞ seperator2
```

3. Validate data model

```
Name
⊟ ☒ GIFHEADER
      Signature
      Ver
  ⊟ ☒ LOG                PTOR
        W
        H
      ☒ S
```

4. Generate fuzzed inputs

5. Send fuzzed input to app

6. Monitor target app

7. Analyze logs

http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

Fuzzing: key characteristics

1. More or less random modification of inputs
2. Monitoring of target application
3. Huge amount of inputs for target are send
4. Automated and repeatable

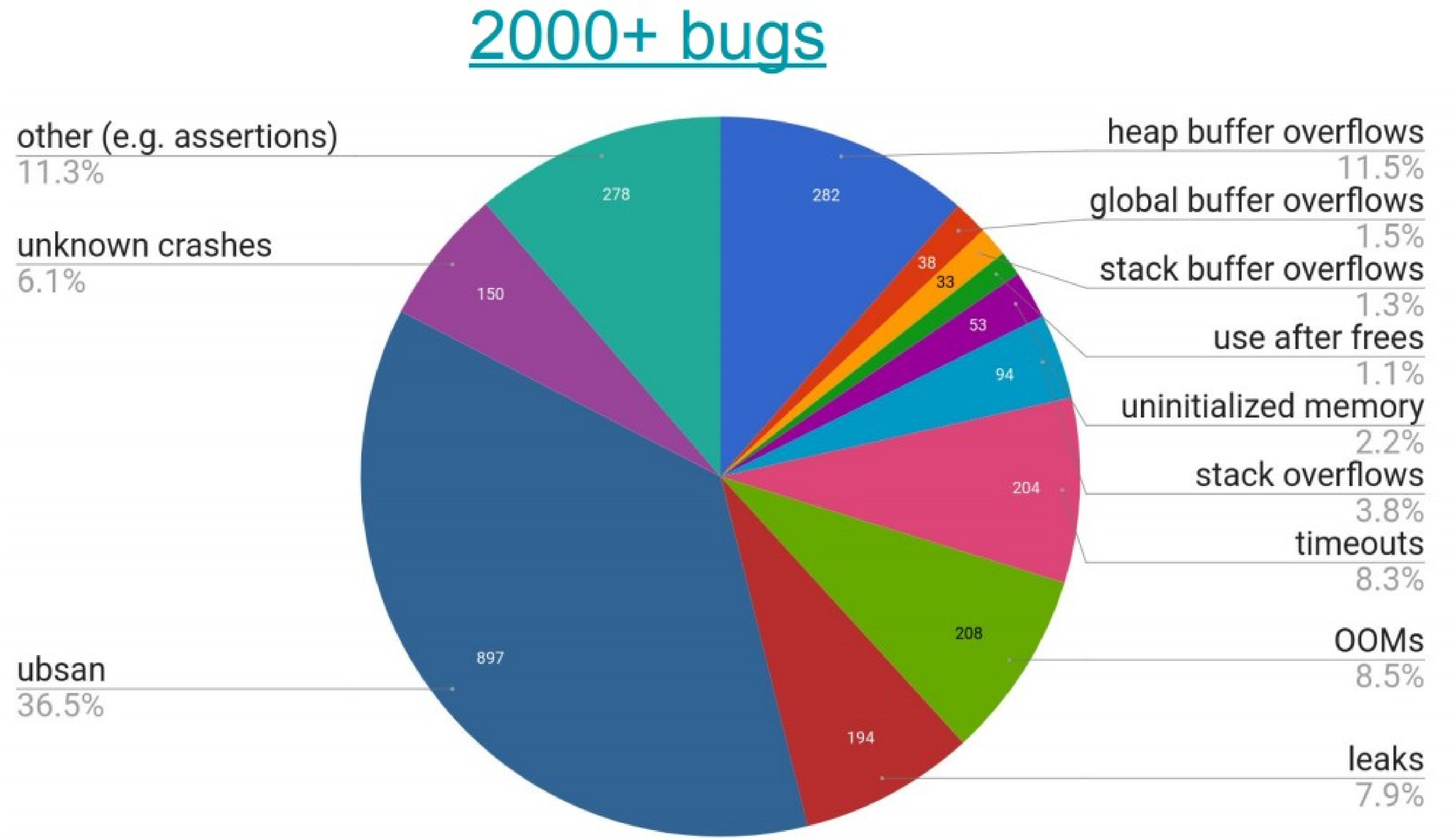
Fuzzing - advantages/disadvantages

- Fuzzing advantages ➕
  - Very simple design
  - Allow to find bugs missed by human eye
  - Sometimes the only way to test (closed system)
  - Repeatable (crash inputs stored)
- Fuzzing disadvantages ➖
  - Usually simpler bugs found (low hanging fruit)
  - Increased difficulty to evaluate impact or dangerosity
  - Closed system is often evaluated, black box testing

What kind of bugs is usually found?

- Memory corruption bugs (buffer overflows...)
- Parser bugs (crash of parser on malformed input)
- Invalid error handling (other then expected error)
- Threading errors (requires sufficient setup)
- Correctness bugs (reference vs. new implementation)

Google's OSS-Fuzz



https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_serebryany.pdf

Microsoft VulnScan

- *“Over a 10-month period where VulnScan was used to triage all memory corruption issues for Microsoft Edge, Microsoft Internet Explorer and Microsoft Office products. It had a success rate around 85%, saving an estimated 500 hours of engineering time for MSRC engineers.”*
- https://blogs.technet.microsoft.com/srd/2017/10/03/vulnscan-automated-triage-and-root-cause-analysis-of-memory-corruption-issues/

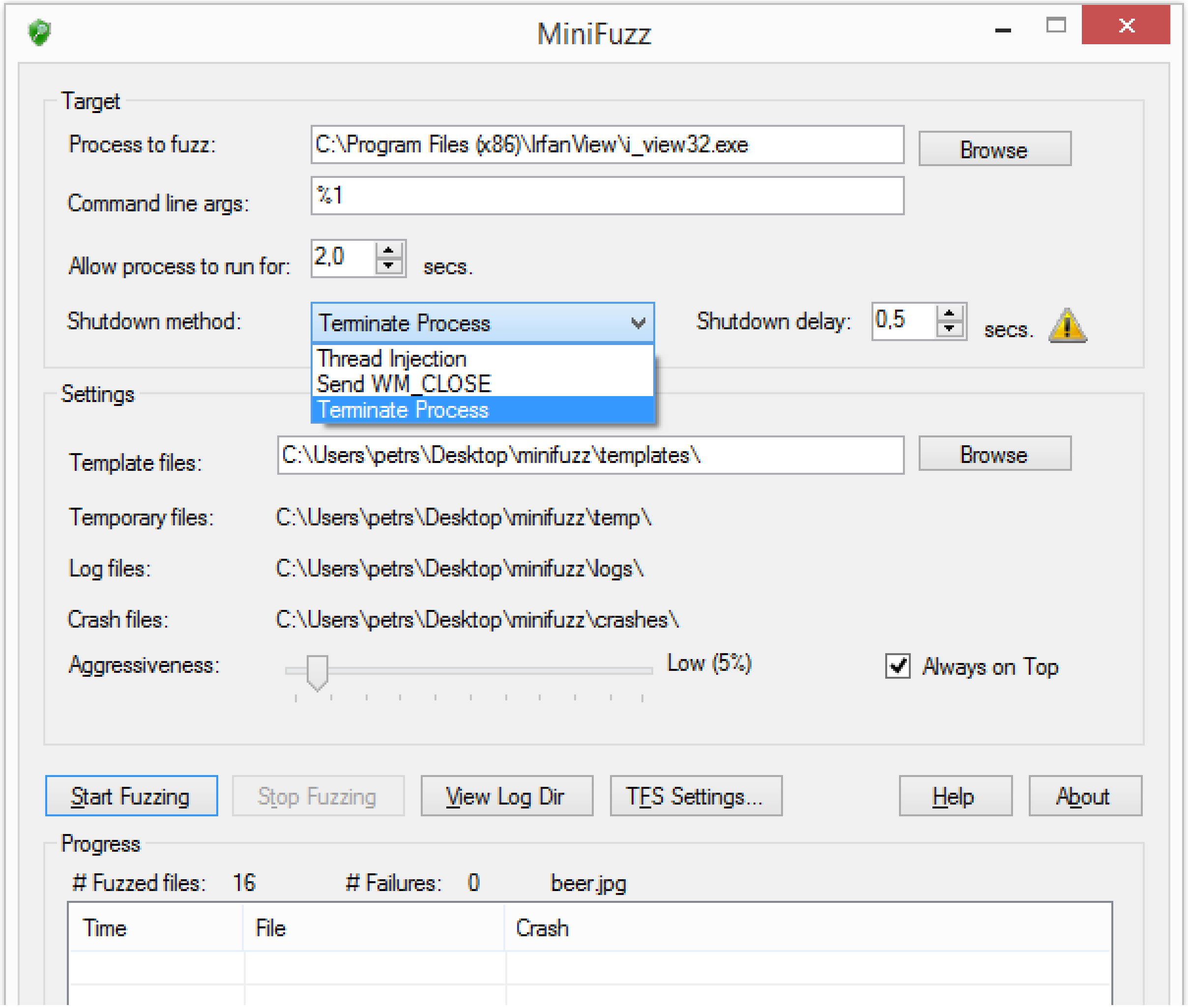
What kind of bugs are usually missed?

- Bugs after input validation (if not modeled properly)
- High-level / architecture bugs (e.g. weak crypto)
- Usability bugs
- …

What kind of applications can be fuzzed?

- Any application/module with an input
  - (sometimes even without inputs, e.g., fault induction)
- Custom (“DIY”) fuzzer
  - Usually full knowledge about target app
  - Kind of randomized integration test (but still repeatable!)
- File fuzzer – input via files
- Network fuzzer – input received via network
- General fuzzing framework
  - Preprepared tools and functions for common tasks (file, packet…)
  - Custom plugins, pre-prepared and custom data models

Microsoft's SDL MiniFuzz File Fuzzer



MiniFuzz

Target

Process to fuzz: C:\Program Files (x86)\IrfanView\i_view32.exe

Command line args: %1

Allow process to run for: 2,0 secs.

Shutdown method: Terminate Process

Thread Injection
Send WM_CLOSE
Terminate Process

Shutdown delay: 0,5 secs.

Settings

Template files: C:\Users\petrs\Desktop\minifuzz\templates\

Temporary files: C:\Users\petrs\Desktop\minifuzz\temp\

Log files: C:\Users\petrs\Desktop\minifuzz\logs\

Crash files: C:\Users\petrs\Desktop\minifuzz\crashes\

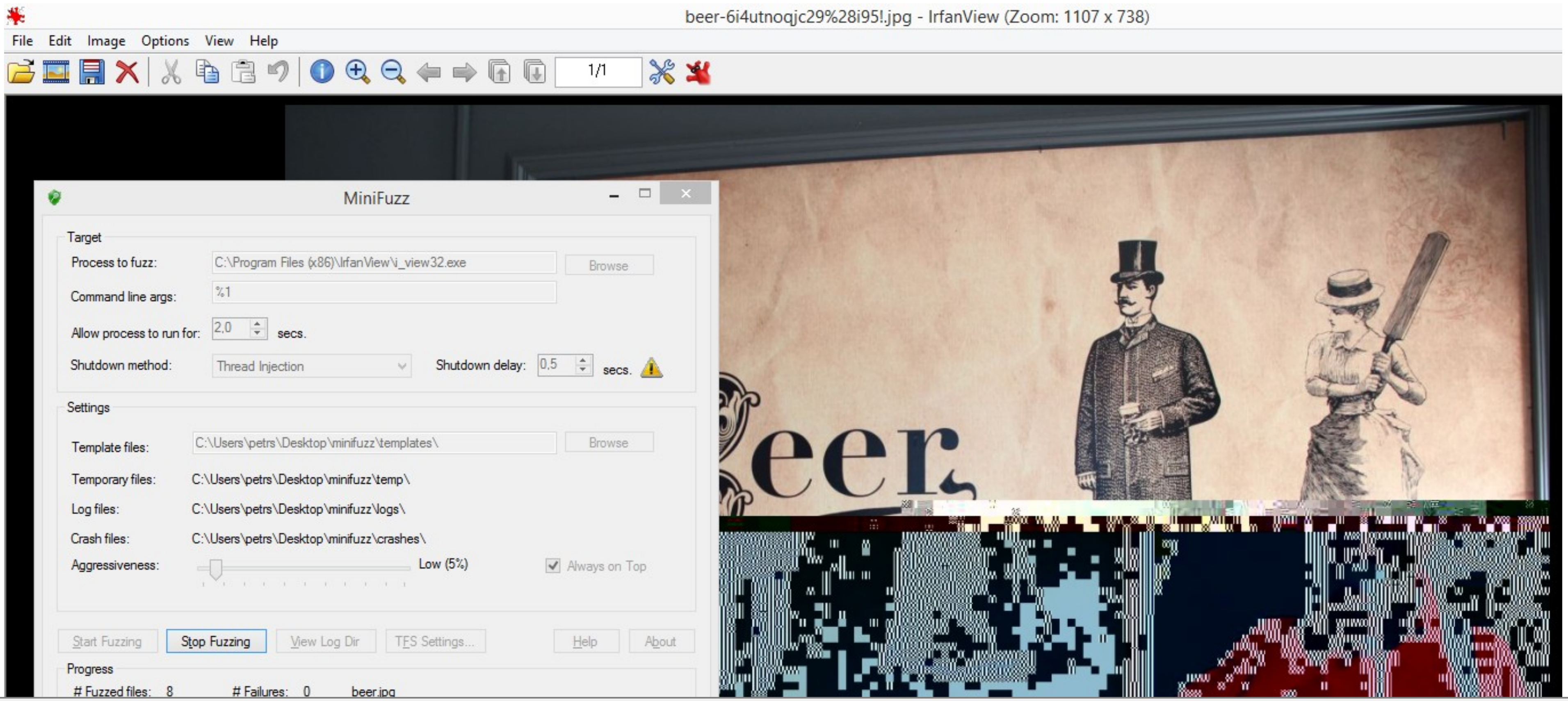
Aggressiveness: Low (5%)

☑ Always on Top

Progress

Fuzzed files: 16 # Failures: 0 beer.jpg

Time	File	Crash



beer-6i4utnoqjc29%28i95!.jpg - IrfanView (Zoom: 1107 x 738)

MiniFuzz

Target

Process to fuzz: C:\Program Files (x86)\IrfanView\i_view32.exe

Command line args: %1

Allow process to run for: 2,0 secs.

Shutdown method: Thread Injection

Shutdown delay: 0,5 secs.

Settings

Template files: C:\Users\petrs\Desktop\minifuzz\templates\

Temporary files: C:\Users\petrs\Desktop\minifuzz\temp\

Log files: C:\Users\petrs\Desktop\minifuzz\logs\

Crash files: C:\Users\petrs\Desktop\minifuzz\crashes\

Aggressiveness: Low (5%)

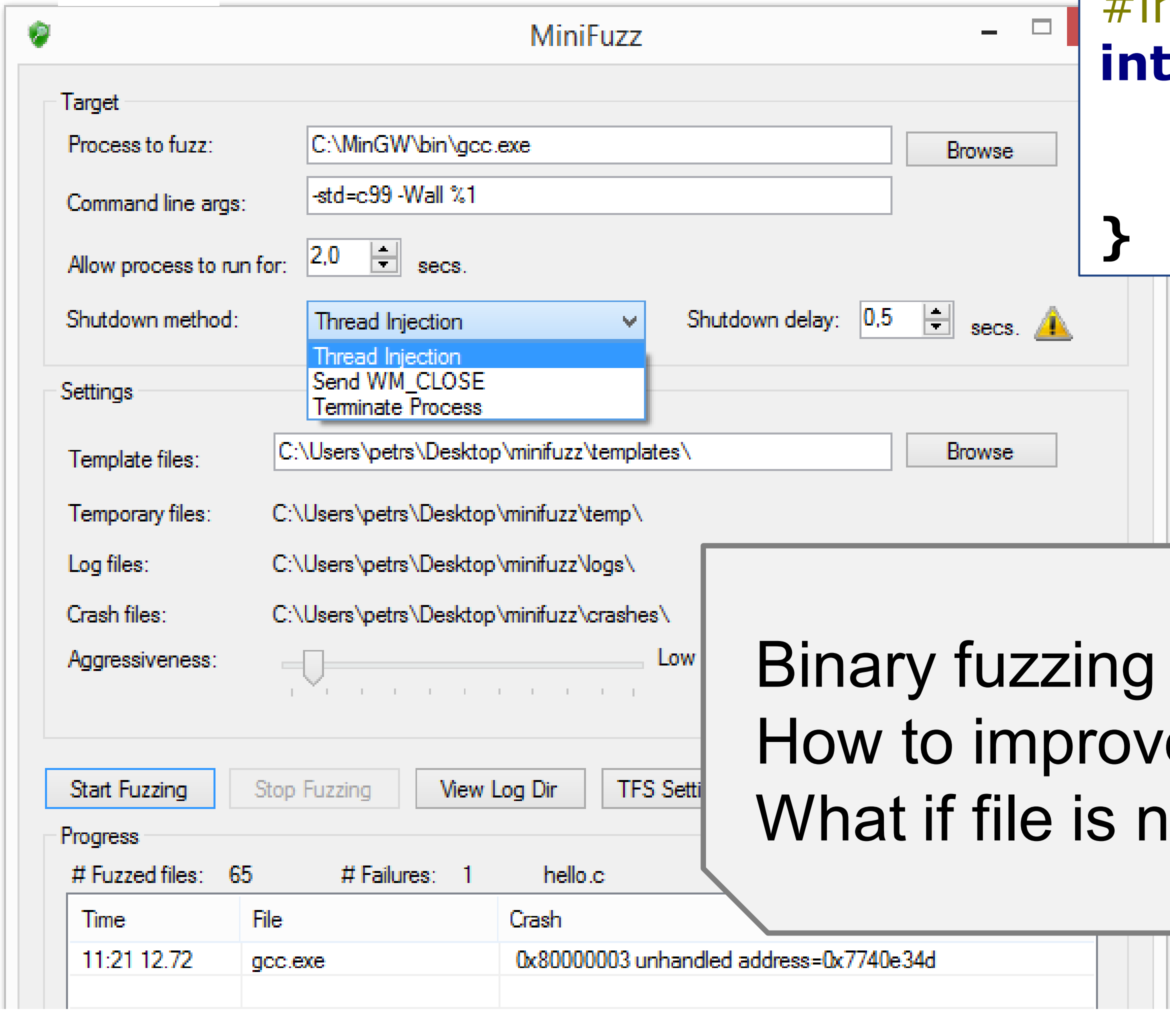
☑ Always on Top

Progress

Fuzzed files: 8 # Failures: 0 beer.jpg

```
<?xml version="1.0"?>
<failures>
  <failure type="Exception Event:Tid=8504, 0x80000003, unhandled, address=0x7740e34d" datetime="11:21:12 12.
    <registers RAX="00000000" RBX="00000000" RCX="7FFF5FC5180A" RDX="00000000" RSI="00000000" RDI="000
    <process name="C:\Program Files (x86)\IrfanView\i_view32.exe" />
    <file name="-std=c99 -Wall C:\minifuzz\temp\beer-0rsw9!h2jf.jpg" />
  </failure>
</failures>
```

MiniFuzz: gcc fuzzing



MiniFuzz

Target

Process to fuzz: C:\MinGW\bin\gcc.exe

Command line args: -std=c99 -Wall %1

Allow process to run for: 2,0 secs.

Shutdown method: Thread Injection

Shutdown delay: 0,5 secs.

Settings

Template files: C:\Users\petrs\Desktop\minifuzz\templates\

Temporary files: C:\Users\petrs\Desktop\minifuzz\temp\

Log files: C:\Users\petrs\Desktop\minifuzz\logs\

Crash files: C:\Users\petrs\Desktop\minifuzz\crashes\

Aggressiveness: Low

Progress

Fuzzed files: 65 # Failures: 1 hello.c

Time	File	Crash
11:21 12.72	gcc.exe	0x80000003 unhandled address=0x7740e34d

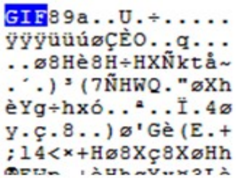
```c
#include<stdio.h>
int main() {
    printf("Hello Fuzzy World");
    return 0;
}
```

Binary fuzzing of source code???

How to improve test coverage?

What if file is not an input?

1. Investigate app in/out



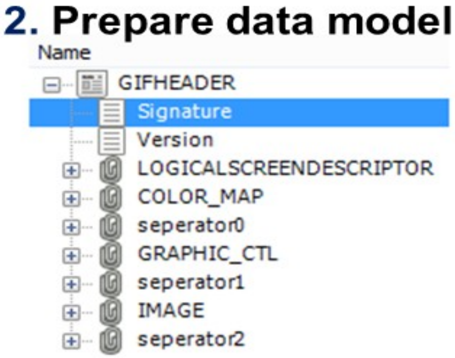
```
GIF89a..U.÷.....
ÿÿÿüüúøÇÈO..q...
..ø8Hè8H÷HXÑktå~
.´.)³(7ÑHWQ."øXh
èYg÷hxó..ª..Ï.4ø
y.ç.8..)ø'Gè(E.+
;14<×+Hø8Xç8XøHh
```

http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

INVESTIGATE APPLICATION

What kind of inputs and strategy?

- Type of inputs?
  - File, network packets, structure, data model, state(-less)
- What environment setup is necessary?
  - Fuzzing on live system?
  - Multiple entities inside VMs? Networking?
- Isolated vs. cooperating components?
  - We don’t like to mock everything
- What tools are readily available?



2. Prepare data model

Name

- GIFHEADER
  - Signature
  - Version
  - LOGICALSCREENDESCRIPTOR
  - COLOR_MAP
  - seperator0
  - GRAPHIC_CTL
  - seperator1
  - IMAGE
  - seperator2

http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

MODELLING

Input preparation

- *Time intensive part of fuzzing (if model !exists yet)*

1. Fully random data
2. Random modification of valid input
3. Modification of valid input with fuzz vectors
4. Modification of valid input with mutator
5. Fuzzing via intermediate proxy

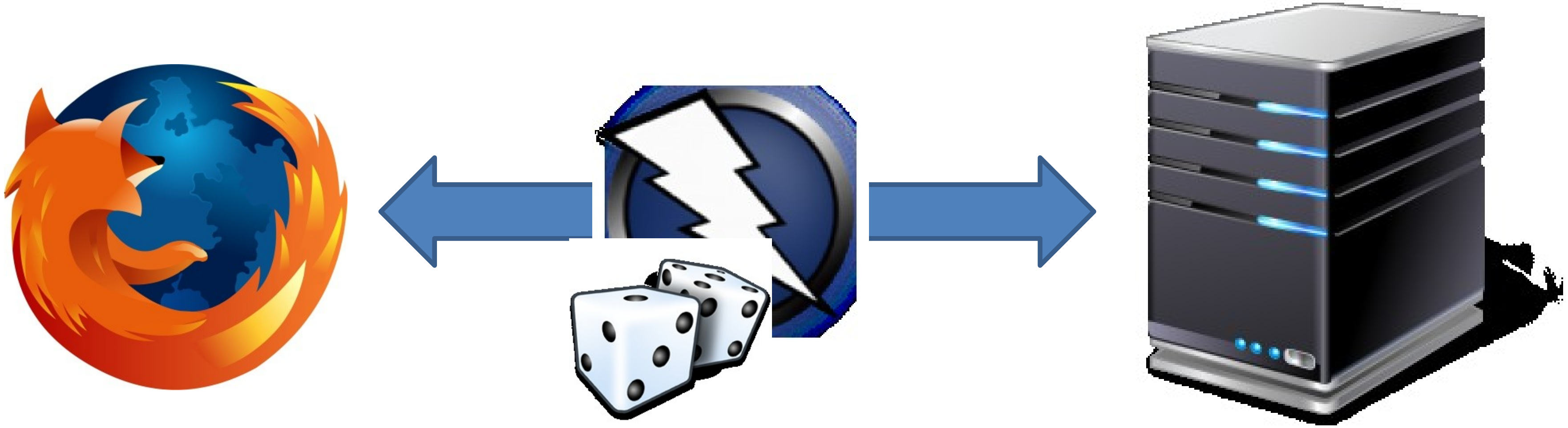
Radamsa fuzzer

- *"…easy-to-set-up general purpose shotgun test to expose the easiest cracks…"*
  – https://code.google.com/p/ouspg/wiki/Radamsa
- Just provide input files, all other settings automatic
  – **cat** file **|** radamsa **>** **file.fuzzed**

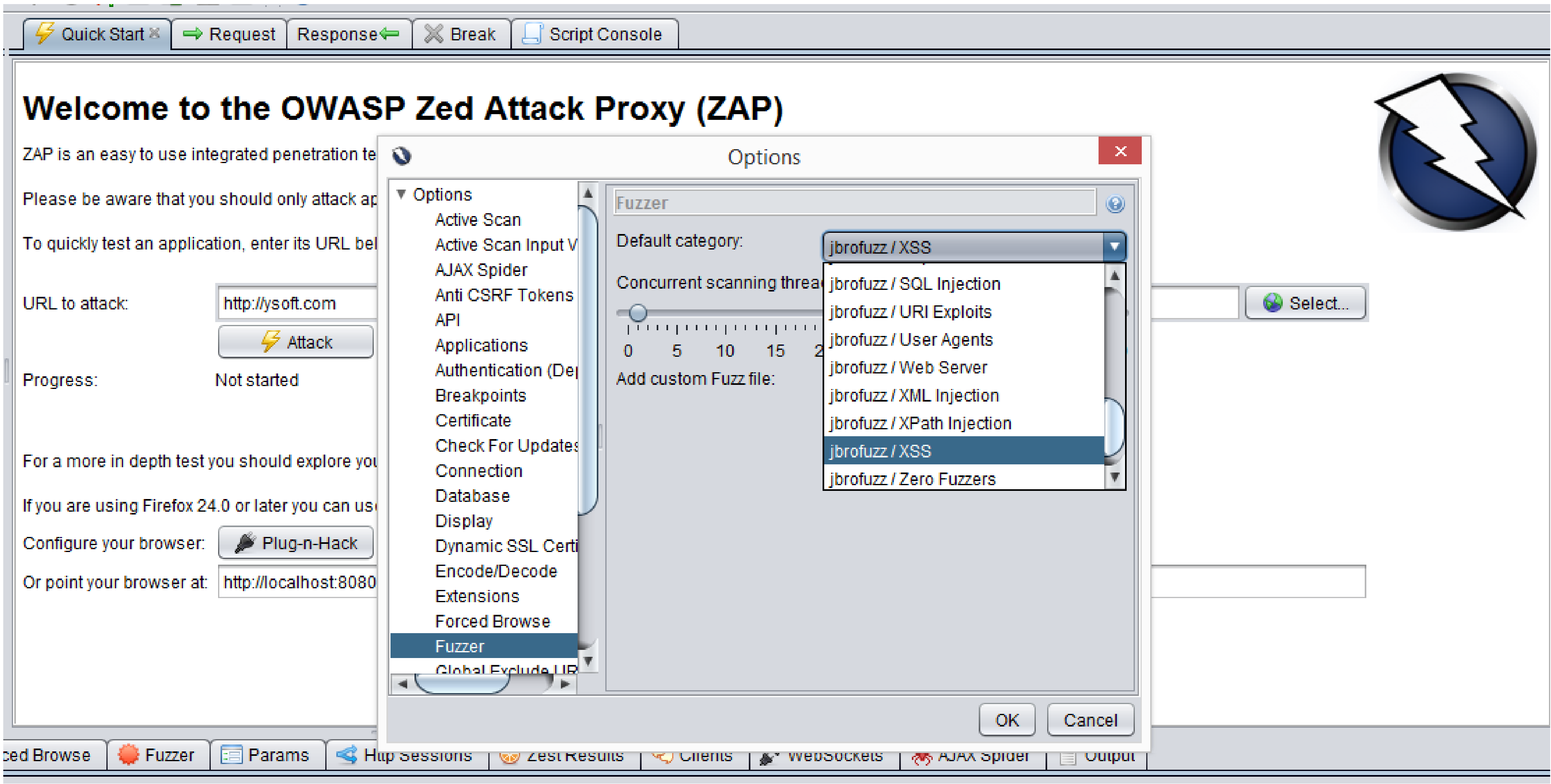
```
>echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
 1 + (2 + (2 + (3 + 4?)
 1 + (2 + (3 +?4))
 18446744073709551615 + 4)))
 1 + (2 + (3 + 170141183460469231731687303715884105727))
```

Fuzzing via intermediate proxy

- Fuzzer modifies valid flow according to data model
- Usually used for fuzzing of state-full protocols
  - Modelling states and interactions would be difficult
  - Target application(s) takes care of states and valid input



OWASP’s ZAP – fuzz strategy settings



Quick Start | Request | Response | Break | Script Console

Welcome to the OWASP Zed Attack Proxy (ZAP)

ZAP is an easy to use integrated penetration te

Please be aware that you should only attack ap

To quickly test an application, enter its URL bel

URL to attack: http://ysoft.com

Attack

Progress: Not started

For a more in depth test you should explore yo

If you are using Firefox 24.0 or later you can us

Configure your browser: Plug-n-Hack

Or point your browser at: http://localhost:8080

Options

Fuzzer

Default category: jbrofuzz / XSS

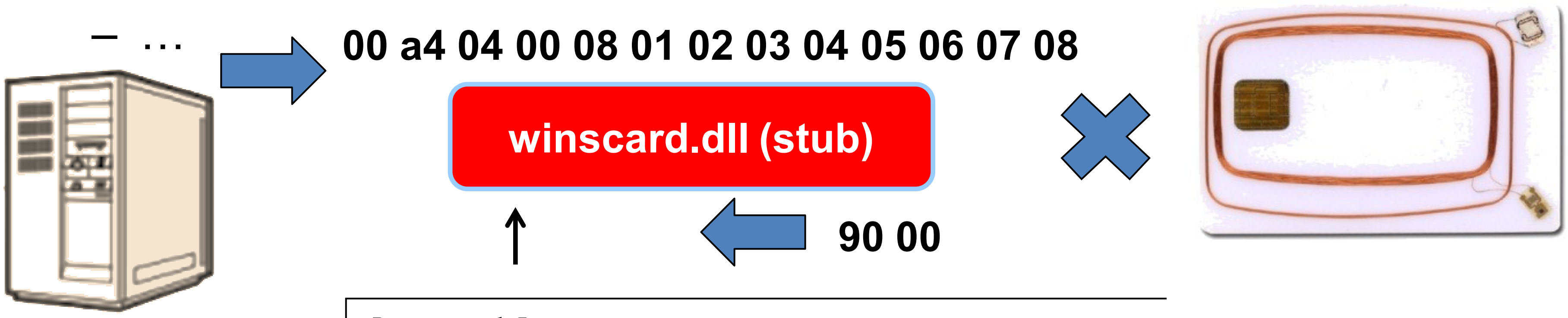
Concurrent scanning threa

Add custom Fuzz file:

http://www.fi.muni.cz/~xsvenda/apduinspect.html

APDUPlay - Smart card fuzzing

- Host to smart card communication done via PC/SC
- Custom winscard.dll stub written
- Manipulate incoming/outgoing APDUs
  - modify packet content
  - replay of previous packets
  - …



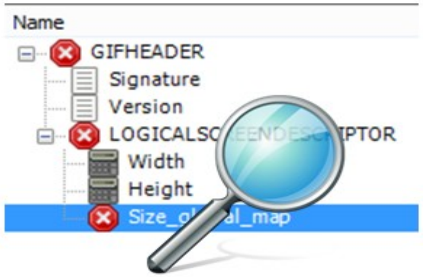
00 a4 04 00 08 01 02 03 04 05 06 07 08

winscard.dll (stub)

90 00

```
[RULE1]
MATCH1=in=1;t=0;cla=00;ins=a4;p1=04;
ACTION=in=0;data0=90 00;le=02;
```

3. Validate data model



Name

GIFHEADER

Signature

Version

LOGICALSCREENDESCRIPTOR

Width

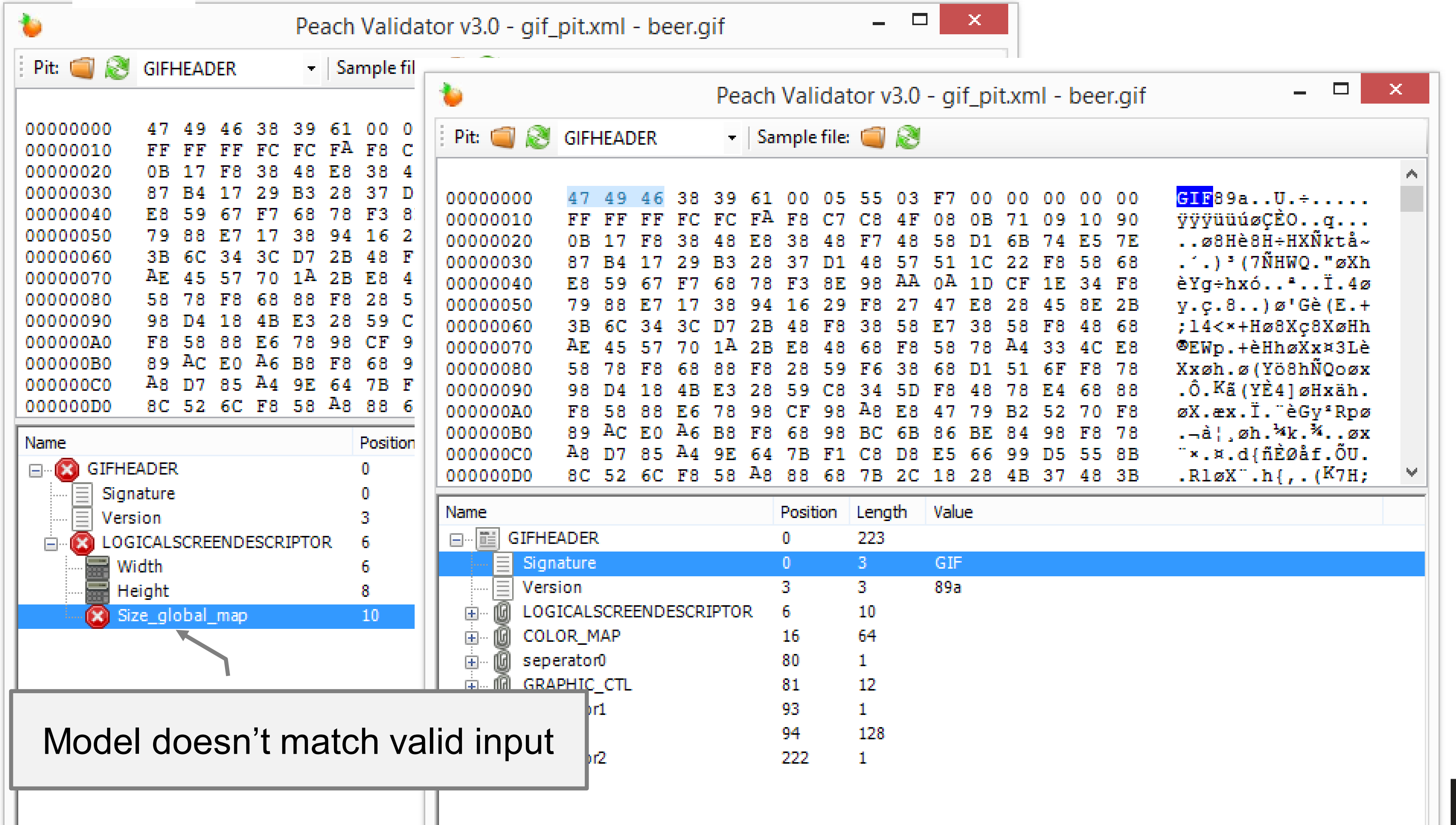
Height

Size_global_map

http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

VALIDATION

Peach Validator 3.0



```
00000000   47 49 46 38 39 61 00 0
00000010   FF FF FF FC FC FA F8 C
00000020   0B 17 F8 38 48 E8 38 4
00000030   87 B4 17 29 B3 28 37 D
00000040   E8 59 67 F7 68 78 F3 8
00000050   79 88 E7 17 38 94 16 2
00000060   3B 6C 34 3C D7 2B 48 F
00000070   AE 45 57 70 1A 2B E8 4
00000080   58 78 F8 68 88 F8 28 5
00000090   98 D4 18 4B E3 28 59 C
000000A0   F8 58 88 E6 78 98 CF 9
000000B0   89 AC E0 A6 B8 F8 68 9
000000C0   A8 D7 85 A4 9E 64 7B F
000000D0   8C 52 6C F8 58 A8 88 6
```

Name	Position
GIFHEADER	0
Signature	0
Version	3
LOGICALSCREENDESCRIPTOR	6
Width	6
Height	8
Size_global_map	10

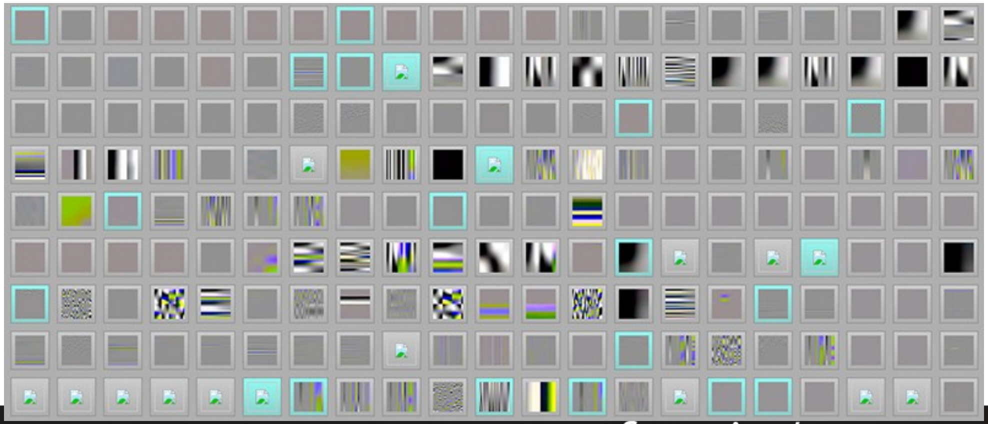
Model doesn't match valid input

```
00000000   47 49 46 38 39 61 00 05 55 03 F7 00 00 00 00 00   GIF89a..U.÷.....
00000010   FF FF FF FC FC FA F8 C7 C8 4F 08 0B 71 09 10 90   ÿÿÿüüúøÇÈO..q...
00000020   0B 17 F8 38 48 E8 38 48 F7 48 58 D1 6B 74 E5 7E   ..ø8Hè8H÷HXÑktå~
00000030   87 B4 17 29 B3 28 37 D1 48 57 51 1C 22 F8 58 68   .´.)³(7ÑHWQ."øXh
00000040   E8 59 67 F7 68 78 F3 8E 98 AA 0A 1D CF 1E 34 F8   èYg÷hxó..ª..Ï.4ø
00000050   79 88 E7 17 38 94 16 29 F8 27 47 E8 28 45 8E 2B   y.ç.8..)ø'Gè(E.+
00000060   3B 6C 34 3C D7 2B 48 F8 38 58 E7 38 58 F8 48 68   ;l4<×+Hø8Xç8XøHh
00000070   AE 45 57 70 1A 2B E8 48 68 F8 58 78 A4 33 4C E8   ®EWp.+èHhøXx¤3Lè
00000080   58 78 F8 68 88 F8 28 59 F6 38 68 D1 51 6F F8 78   Xxøh.ø(Yö8hÑQoøx
00000090   98 D4 18 4B E3 28 59 C8 34 5D F8 48 78 E4 68 88   .Ô.Kã(YÈ4]øHxäh.
000000A0   F8 58 88 E6 78 98 CF 98 A8 E8 47 79 B2 52 70 F8   øX.æx.Ï.¨èGy²Rpø
000000B0   89 AC E0 A6 B8 F8 68 98 BC 6B 86 BE 84 98 F8 78   .¬à¦¸øh.¼k.¾..øx
000000C0   A8 D7 85 A4 9E 64 7B F1 C8 D8 E5 66 99 D5 55 8B   ¨×.¤.d{ñÈØåf.ÕU.
000000D0   8C 52 6C F8 58 A8 88 68 7B 2C 18 28 4B 37 48 3B   .RløX¨.h{,.(K7H;
```

Name	Position	Length	Value
GIFHEADER	0	223	
Signature	0	3	GIF
Version	3	3	89a
LOGICALSCREENDESCRIPTOR	6	10	
COLOR_MAP	16	64	
seperator0	80	1	
GRAPHIC_CTL	81	12	
r1	93	1	
	94	128	
r2	222	1	

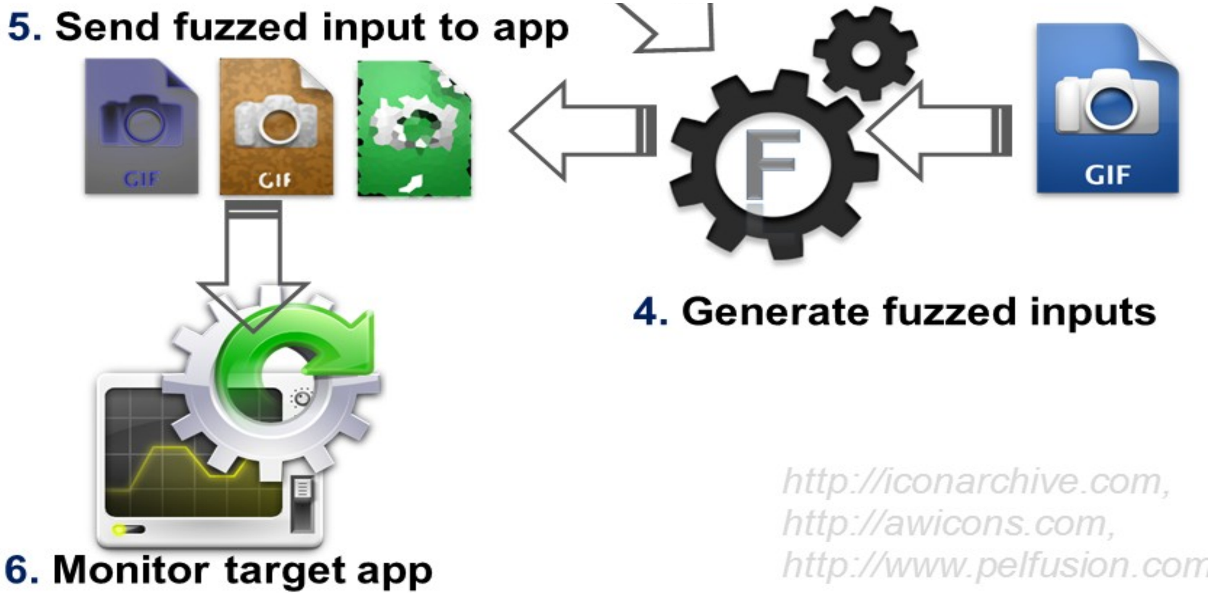
American fuzzy lop

- State of the art and very powerful tool
- High speed fuzzer http://lcamtuf.coredump.cx/afl/
- Sophisticated generation of test cases (coverage)
- Automatic generation of input templates
  - E.g., valid JPEG image from “hello” string after few days
  - http://lcamtuf.blogspot.cz/2014/11/pulling-jpegs-out-of-thin-air.html
- Lots of real bugs found



Test coverage

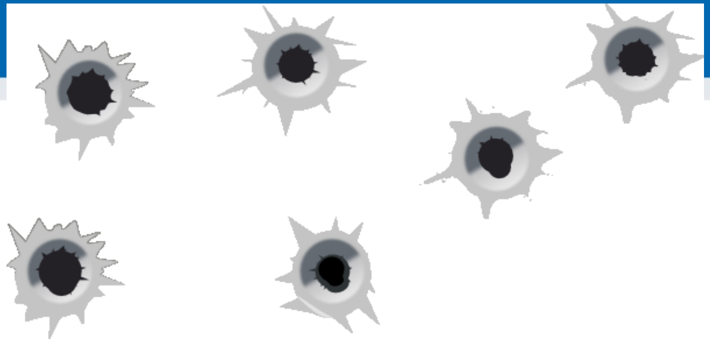
- Random inputs have low coverage (usually)
  - Number of blocks visited in target binary
- Smart fuzzing tries to improve coverage
  - Way how to generate new inputs from existing
- E.g., Peach's minset tool
  - Gather a lot of inputs (files)
  - Run minset tool, traces with coverage stats are collected
  - Minimal set of files to achieve coverage is computed
  - Selected files are used as templates for fuzzing
- E.g. AFL fuzzer uses compile time instrumentation + genetic programming to create test cases



http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

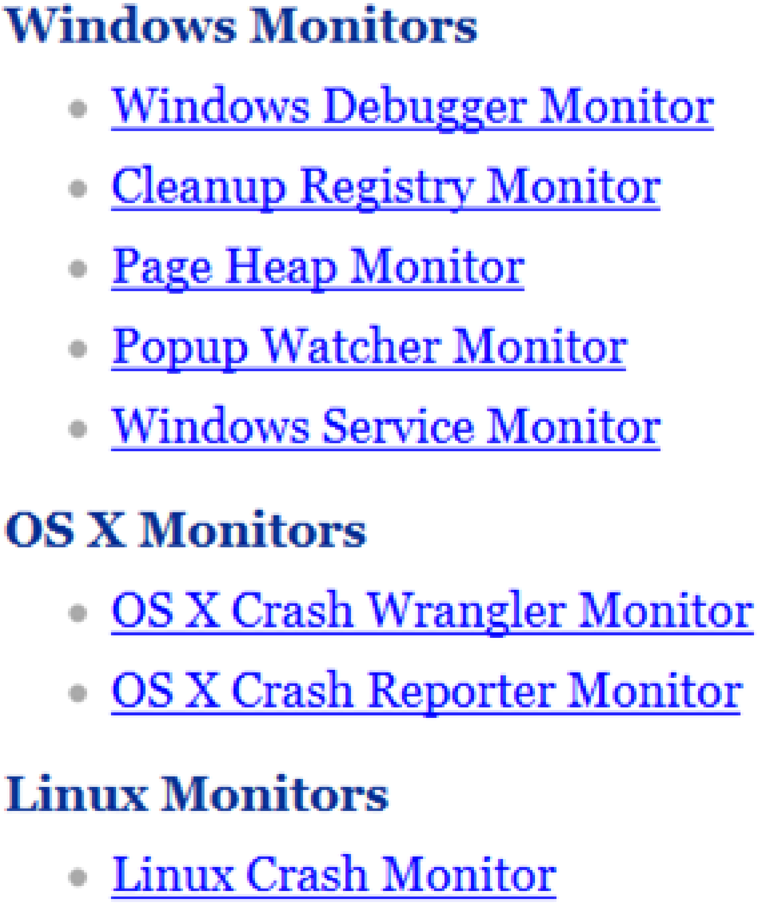
START, GENERATE, MONITOR

How to detect “hit”?



- Application crash, uncaught exception…
  - Clear faults, easy to detect
- Error returned
  - Some errors are valid response
  - Some errors are valid response only in selected states
- Input accepted even when it shouldn't be
  - E.g., packet with incorrect checksum or modified field
- Some operation performed in incorrect state
  - E.g., door open without proper authentication
- Application behavior is impaired
  - E.g., response time significantly increases
- …

Peach monitors



Windows Monitors

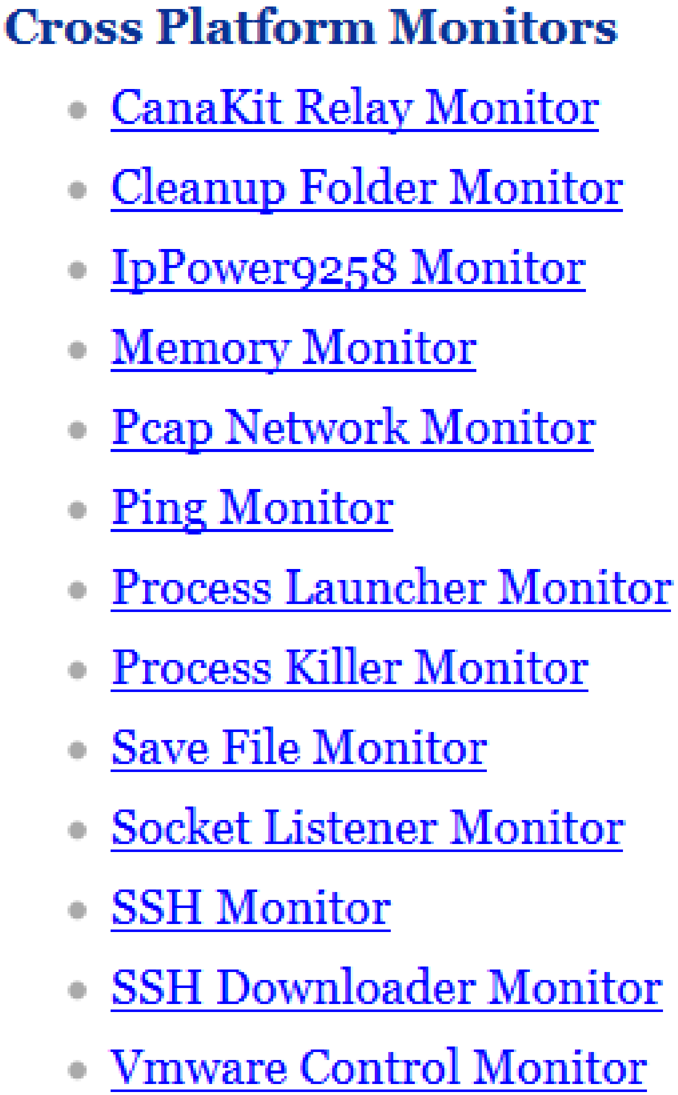
- Windows Debugger Monitor
- Cleanup Registry Monitor
- Page Heap Monitor
- Popup Watcher Monitor
- Windows Service Monitor

OS X Monitors

- OS X Crash Wrangler Monitor
- OS X Crash Reporter Monitor

Linux Monitors

- Linux Crash Monitor



Cross Platform Monitors

- CanaKit Relay Monitor
- Cleanup Folder Monitor
- IpPower9258 Monitor
- Memory Monitor
- Pcap Network Monitor
- Ping Monitor
- Process Launcher Monitor
- Process Killer Monitor
- Save File Monitor
- Socket Listener Monitor
- SSH Monitor
- SSH Downloader Monitor
- Vmware Control Monitor

7. Analyze logs

http://iconarchive.com,
http://awicons.com,
http://www.pelfusion.com

ANALYZE

What to do with hit results?

- *Time intensive part of fuzzing*
- Not all hits are relevant (at least at the beginning)
  - Crashes by values not controllable by an attacker
  - !exploitable https://msecdbg.codeplex.com/
- Hits reproduction
  - Hit can be result of cumulative series of operations
- Many hits are duplicates
  - Inputs are different, but hit caused in the same part of code
- (Automatic) Bucketing of hits
  - E.g., Peach performs bucking based on signature of callstack

Summary for fuzzing

- Fuzzers are cheap way to detect simpler bugs
  - If you don’t use it, others will
- Try to find tool that fits your particular scenario
  - Check activity of development, support
- Fuzzing frameworks can ease variety of setups
  - But bit steeper learning curve
- If fuzzing will not find any bugs, check your model
- Try it!

Fuzzing driven development (FDD)

- Test-driven development (TDD)
  - Write tests first, only later implement functionality
  - Will result in testable code (smaller functions, well defined)
- Fuzzing driven development (FDD)
  - Continuous fuzzing of an application
  - Structure application to enable and support fuzzing
  - Will result in “fuzzable” code (deep penetration into app)
- Google OSS-Fuzz
  - Large-scale continuous fuzzing of important open-source projects on Google’s servers
  - Can be replicated in your Continuous Integration server

TAINT ANALYSIS

Taint analysis

- Form of flow analysis
- Follow propagation of sensitive values inside program
  - e.g., user input that can be manipulated by an attacker
  - find all parts of program where value can “reach”
- *“Information* **flows** *from object x to object y, denoted x→y , whenever information stored in x is transferred to, object y.” D. Denning*
- Native support in some languages (Ruby, Perl)
  - But not C++/Java ☹, FindSecurityBugs adds taint for Java

Taint sources

- Files (*.pdf, *.doc, *.js, *.mp3...)
- User input (keyboard, mouse, touchscreen)
- Network traffic
- USB devices
- ...

- Every time there is information flow from value from untrusted source to other object X, object X is *tainted*
  - labeled as “tainted”

Conclusions

- Dynamic analyzers can profile application
  - and find bugs not found by static analysis
- Fuzzing is “cheap” blackbox approach via malformed inputs
- Mandatory reading
  - Kostya Serebryany, OSS-Fuzz Google's continuous fuzzing service for open source software
  - https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_serebryany.pdf

Questions

EXTENSION SLIDES – FOR IGNITED ☺

References

- Some books available, but…
- Michael Eddington, Demystifying fuzzers
  - Comparison of open-source tools, cost of adoption
  - BlackHat 2009, https://www.blackhat.com/presentations/bh-usa-09/EDDINGTON/BHUSA09-Eddington-DemystFuzzers-PAPER.pdf
  - https://www.blackhat.com/presentations/bh-usa-09/EDDINGTON/BHUSA09-Eddington-DemystFuzzers-SLIDES.pdf
  - RSA Conference 2010 talk https://www.youtube.com/watch?v=Bm3MfndrI1Y
- OWASP fuzzing guidelines
  - https://www.owasp.org/index.php/Fuzzing
  - https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors
- Tutorials and research papers on fuzzing http://fuzzing.info/papers/

Peach tutorials

- Basic usage against vulnserver
  - http://rockfishsec.blogspot.ch/2014/01/fuzzing-vulnserver-with-peach-3.html
- Advanced tutorial (ZIP format fuzzing) – very good
  - http://www.flinkd.org/2011/07/fuzzing-with-peach-part-1/
- Tutorial for RAR fuzzing
  - http://www.flinkd.org/2011/11/fuzzing-with-peach-part-2-fixups-2/

References

- MS post on Test coverage by fuzzing
  - http://blogs.technet.com/b/srd/archive/2010/02/24/using-code-coverage-to-improve-fuzzing-results.aspx
- Application and file fuzzing
  - http://resources.infosecinstitute.com/application-and-file-fuzzing/
- How I Learned to Stop Fuzzing and Find More Bugs
  - https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-west.pdf

DYNAMIC ANALYSIS - PROFILING (WHITEBOX)

Automatic measurement - profiling

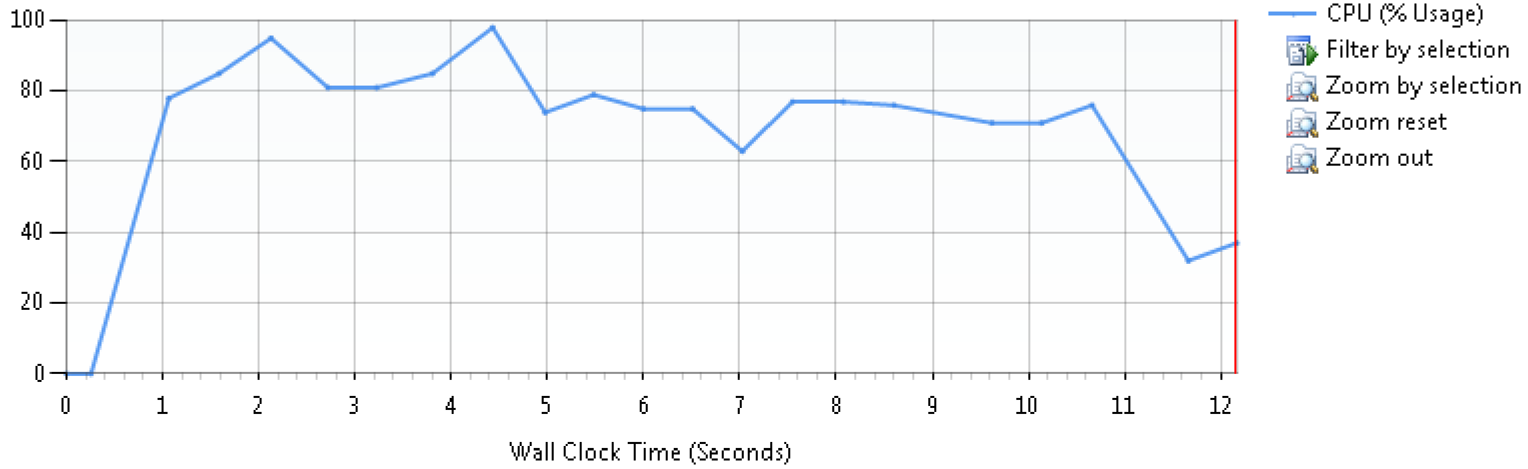
- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
  - what function called actual one
  - based on real code execution (condition jumps)
- Many other statistics, depend on the tools
- Helps to focus and scope security analysis

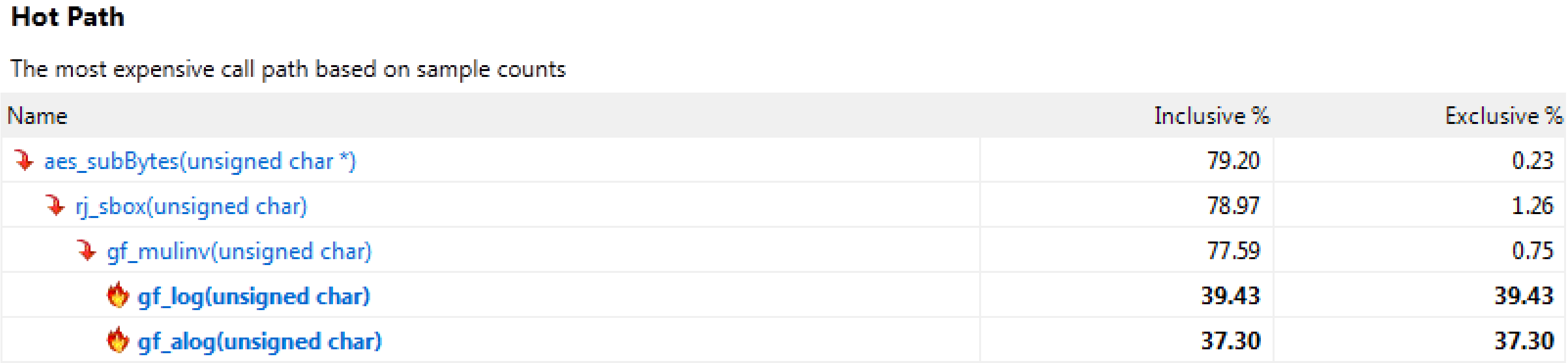
MS Visual Studio Profiler

- Analyze→Launch Performance Wizard
- Profiling method: CPU Sampling
  - – check periodically what is executed on CPU
  - – accurate, low overhead
- Profiling method: Instrumentation
  - – automatically inserts special accounting code
  - – will return exact function call counter
  - – (may affect performance timings a bit)
    - additional code present
- May require admin privileges (will ask)

MS VS Profiler – results (Summary)

- Where to start the optimization work?





Hot Path

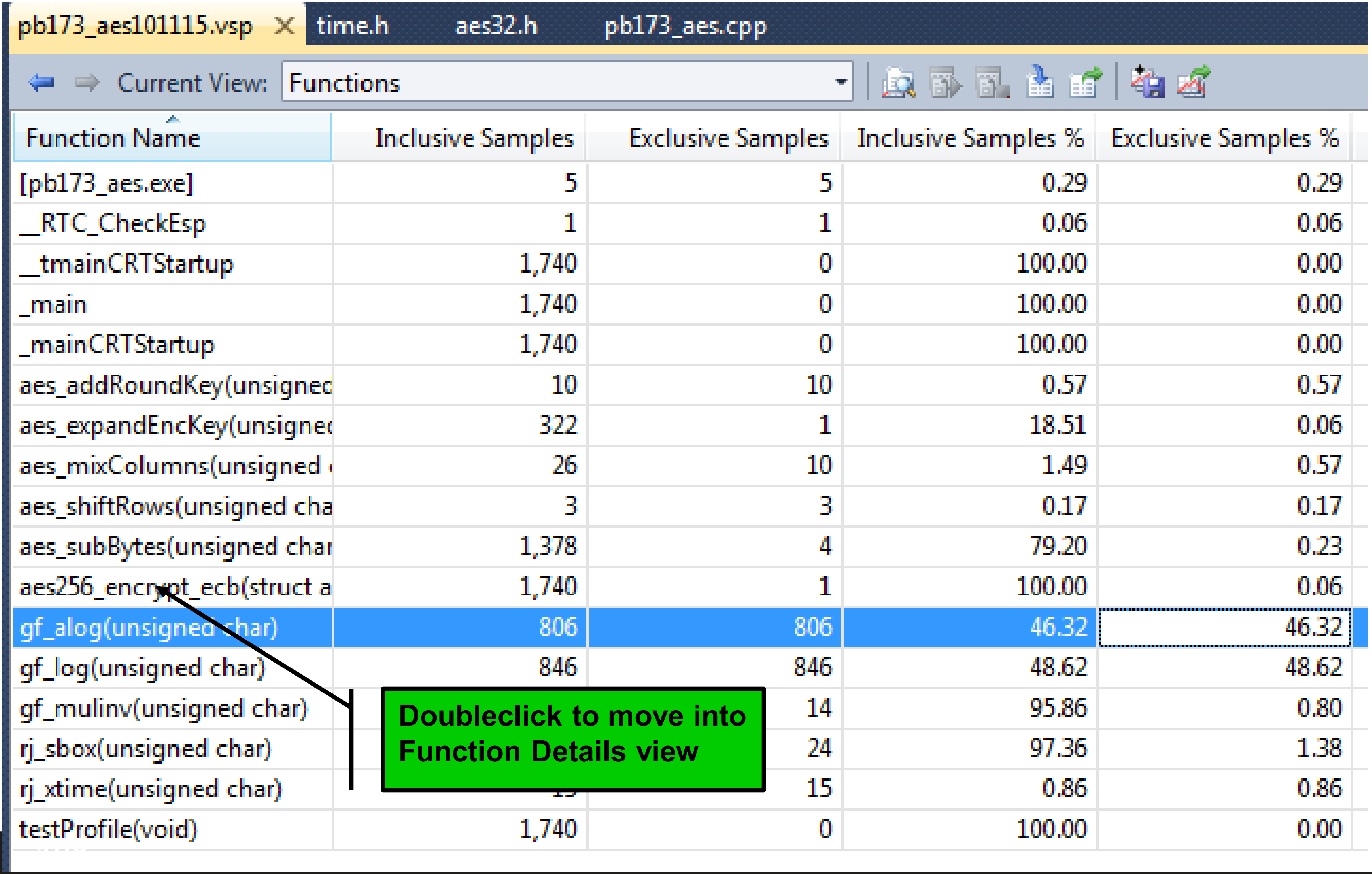
The most expensive call path based on sample counts

Name	Inclusive %	Exclusive %
aes_subBytes(unsigned char *)	79.20	0.23
rj_sbox(unsigned char)	78.97	1.26
gf_mulinv(unsigned char)	77.59	0.75
gf_log(unsigned char)	**39.43**	**39.43**
gf_alog(unsigned char)	**37.30**	**37.30**

MS VS Profiler – results (Functions)

- Result given in number of sampling hits
  - meaningful result is % of total time spend in function
- Inclusive sampling
  - samples hit in function or its children
  - aggregate over call stack for given function
- Exclusive sampling
  - samples hit in exclusively in given function
  - usually what you want
    - fraction of time spend in function code (not in subfunctions)

MS VS Profiler – results (Functions)



Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
[pb173_aes.exe]	5	5	0.29	0.29
_RTC_CheckEsp	1	1	0.06	0.06
_tmainCRTStartup	1,740	0	100.00	0.00
_main	1,740	0	100.00	0.00
_mainCRTStartup	1,740	0	100.00	0.00
aes_addRoundKey(unsignec	10	10	0.57	0.57
aes_expandEncKey(unsignec	322	1	18.51	0.06
aes_mixColumns(unsigned	26	10	1.49	0.57
aes_shiftRows(unsigned cha	3	3	0.17	0.17
aes_subBytes(unsigned char	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct a	1,740	1	100.00	0.06
gf_alog(unsigned char)	806	806	46.32	46.32
gf_log(unsigned char)	846	846	48.62	48.62
gf_mulinv(unsigned char)	[illegible]	14	95.86	0.80
rj_sbox(unsigned char)	[illegible]	24	97.36	1.38
rj_xtime(unsigned char)	15	15	0.86	0.86
testProfile(void)	1,740	0	100.00	0.00

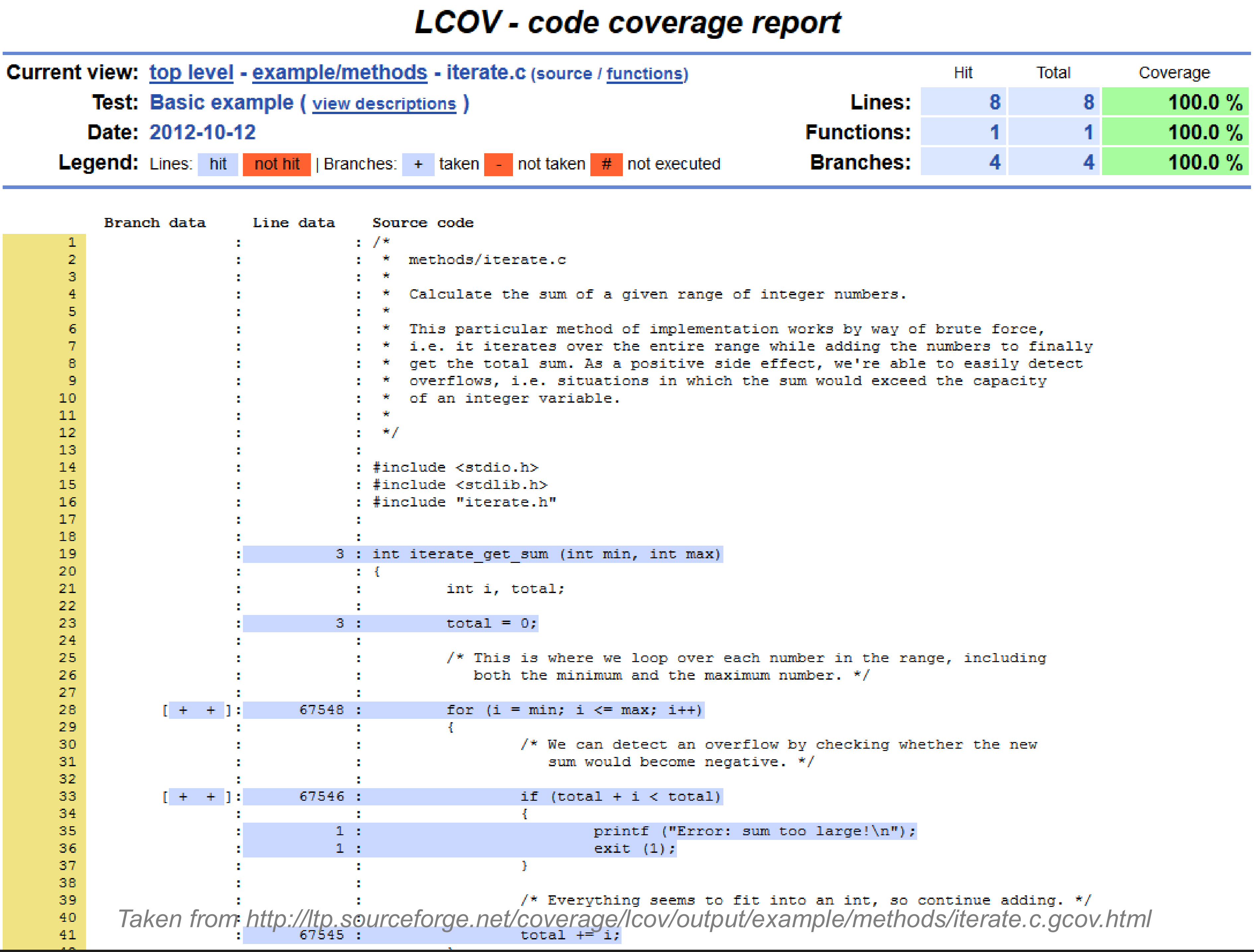
Doubleclick to move into Function Details view

GCC `gcov` tool

- http://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov

1. Compile program by GCC with additional flags
   - gcc -Wall -fprofile-arcs -ftest-coverage main.c
   - gcc -Wall --coverage main.c
   - additional monitoring code is added to binary
2. Execute program
   - files with “.bb" ".bbg" and ".da" extension are created
3. Analyze resulting files with **`gcov`**
   - gcov main.c
   - annotated source code is created

- Lcov - graphical front-end for gcov
  - http://ltp.sourceforge.net/coverage/lcov.php



LCOV - code coverage report

Current view: top level - example/methods - iterate.c (source / functions)

Test: Basic example (view descriptions)

Date: 2012-10-12

Legend: Lines: hit not hit | Branches: + taken - not taken # not executed

	Hit	Total	Coverage
Lines:	8	8	100.0 %
Functions:	1	1	100.0 %
Branches:	4	4	100.0 %

```
       Branch data     Line data    Source code

                  :              : /*
                  :              :  *  methods/iterate.c
                  :              :  *
                  :              :  *  Calculate the sum of a given range of integer numbers.
                  :              :  *
                  :              :  *  This particular method of implementation works by way of brute force,
                  :              :  *  i.e. it iterates over the entire range while adding the numbers to finally
                  :              :  *  get the total sum. As a positive side effect, we're able to easily detect
                  :              :  *  overflows, i.e. situations in which the sum would exceed the capacity
                  :              :  *  of an integer variable.
                  :              :  *
                  :              :  */
                  :              :
                  :              : #include <stdio.h>
                  :              : #include <stdlib.h>
                  :              : #include "iterate.h"
                  :              :
                  :              :
                  :            3 : int iterate_get_sum (int min, int max)
                  :              : {
                  :              :         int i, total;
                  :              :
                  :            3 :         total = 0;
                  :              :
                  :              :         /* This is where we loop over each number in the range, including
                  :              :            both the minimum and the maximum number. */
                  :              :
             [ +  + ]:       67548 :         for (i = min; i <= max; i++)
                  :              :         {
                  :              :                 /* We can detect an overflow by checking whether the new
                  :              :                    sum would become negative. */
                  :              :
             [ +  + ]:       67546 :                 if (total + i < total)
                  :              :                 {
                  :            1 :                         printf ("Error: sum too large!\n");
                  :            1 :                         exit (1);
                  :              :                 }
                  :              :
                  :              :                 /* Everything seems to fit into an int, so continue adding. */
                  :              :
                  :        67545 :                 total += i;
```

Taken from http://ltp.sourceforge.net/coverage/lcov/output/example/methods/iterate.c.gcov.html

REVERSE ENGINEERING (BLACKBOX)

Reverse engineering

- Art of discovering principles through analysis of structure, functions and operation
- Legality
  - Own binary without documentation
  - Interoperability
  - Anti-virus research
  - Fair use, education
  - Forensics
- Problem with recent copyright laws
  - even attempt to circumvent is illegal
  - not only selling circumvented content

Disassembler vs. debugger

- Static vs. dynamic code analysis
- Debugger vs. Debugger with advanced modification tools (Visual Studio vs. OllyDbg)
- Assembler vs. bytecode
  - Instruction set
  - Register-based vs. stack-based execution

Lena tutorials

- Nice introduction tutorials for reversing/cracking
- Win32 binary
  - Lena tutorials 1 and 2
- Name of the registers
  - (EAX 32bit, AX 16bit, AH/AL 8bit)
- Registers (FPU):
  - Z – zero flag, C – carry flag, S – sign flag
  - EIP ... next address to execute (instruction pointer)
  - EBX ... usually loop counter

Startup resources

- The Reverse Code Engineering Community: http://www.reverse-engineering.net/
- Tutorials for You: http://www.tuts4you.com
- RE on Wikipedia: http://en.wikipedia.org/wiki/Reverse_engineering

Disassembling binary code

- Interactive Disassembler is legendary full-fledged disassembler with ability to disassemble many different platforms.
  - Free version available for non-commercial uses
  - Free version disassemble only Windows binaries
  - http://www.hex-rays.com/idapro/idadownfreeware.htm
- Very nice visualization and debugger feature (similar as OllyDbg)
  - Try it!

Decompiling binary code

- Decompiler is able to produce source code from binary code. Decompiler needs to do disassembling first and then try to create code that will in turn produce binary code you have at the beginning.
- Resulting code will NOT contain information removed during compilation
  - (comments, function names, formatting...)
  - Read http://www.debugmode.com/dcompile/ for more info
- Still can be of great help
- Problem to find well working free disassembler
  - http://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompilers

Resources

- The Reverse Code Engineering Community: http://www.reverse-engineering.net/
- Tutorials for You: http://www.tuts4you.com
- Disassembling tutorial http://www.codeproject.com/KB/cpp/reversedisasm.aspx

OLD FUZZING

Example with many possible inputs

- Multiple inputs to application
- Not possible to evaluate manually
  - or done very frequently: UT, TDD, continuous integration
- Sometimes not possible to bruteforce at all
  - to many combinations
  - usual situation!
- Easy to overlook potential problem

Fuzzer - principle

1. Send high number of different inputs into application
   - user interface (UI), command line options, import/export capabilities
2. Inputs generated randomly or according to predefined pattern
   - Protocol/file-format/data-type dependant
   - E.g., input string with different lengths (1MB user name)
   - E.g., valid input with certain percentage of random modifications (jpg file with random changes)
3. Fuzzer monitors application for crash or emitted error
   - Memory corruption, invalid state…
   - When detected, problem is manually inspected

- https://www.owasp.org/index.php/Fuzzing

Fuzzing – advantages/disadvantages

- Fuzzers tend to find simple bugs
  - more protocol-aware fuzzer is, less weird problems will find
- Fuzzing advantages
  - very simple design
  - allow to find bugs missed by human eye
  - sometimes only way to test (completely closed system)
- Fuzzing disadvantages
  - increased difficulty to evaluate impact/dangerosity
    - closed system is evaluated, black box testing

Types of fuzzing

- Application fuzzing
  - generates inputs for application (stdin, memory...)
- Protocol fuzzing
  - manipulation of protocol level
- File format fuzzing
  - generates malformed file samples
  - if program crashes, debug log is created
  - attack against parser layer
  - attack against codec/application layer
  - example: MS04-028 Microsoft's JPEG GDI+ vulnerability
    - http://technet.microsoft.com/en-us/security/bulletin/ms04-028

Fuzzing – approaches

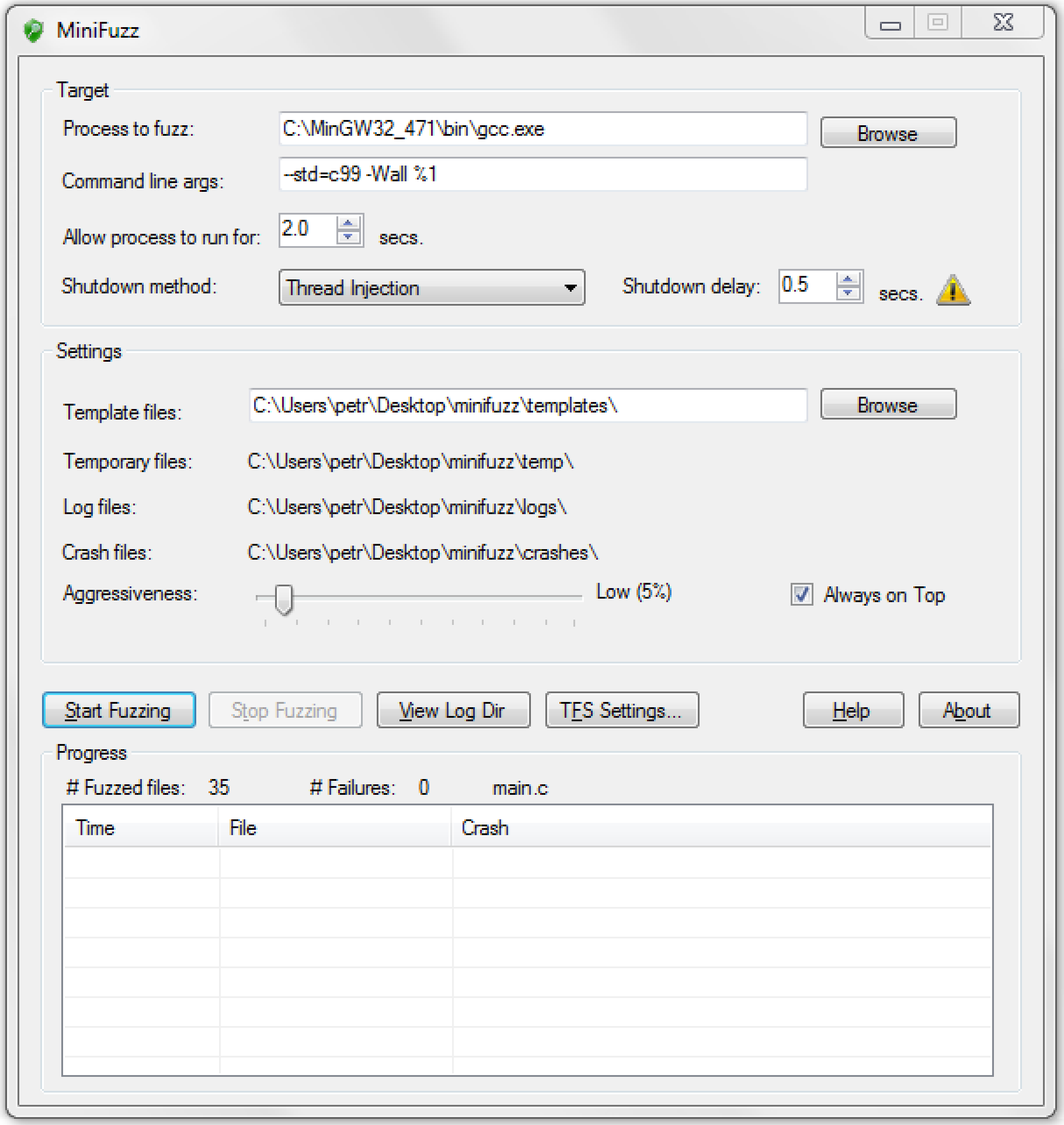
- Capture valid inputs and modify some bytes
  - randomly
  - according to given regular expression
  - Random changes with post-processing (e.g., correct CRC)
- Binary vs. text oriented fuzzing

Available tools

- Microsoft's SDL MiniFuzz File Fuzzer
- Microsoft's SDL Regex Fuzzer
- Ilja van Sprundel's mangle.c
  - https://ext4.wiki.kernel.org/index.php/Filesystem_Testing_Tools/mangle.c
  - filename and header size
  - change between 0 and 10% of header with random bytes
  - example data
- zzuf - multi-purpose fuzzer
  - application input fuzzer
  - intercepting file and network operations and changing random bits in the program's input
  - http://sam.zoy.org/zzuf/

Microsoft’s SDL MiniFuzz File Fuzzer

- Application input files fuzzer
  - http://www.microsoft.com/en-us/download/details.aspx?id=21769
  - UsingMiniFuzz.htm
- Templates for valid input files (multiple)
- Modify valid input file (randomly, % aggressiveness)
- Run application with partially modified inputs
- Log resulting crash (if happen)
  - exception, CPU registers...
- Can be incorporated directly into Visual Studio
- Video overview
  - http://msdn.microsoft.com/en-us/security/gg675011.aspx



MiniFuzz

Target

Process to fuzz: C:\MinGW32_471\bin\gcc.exe

Command line args: --std=c99 -Wall %1

Allow process to run for: 2.0 secs.

Shutdown method: Thread Injection

Shutdown delay: 0.5 secs.

Settings

Template files: C:\Users\petr\Desktop\minifuzz\templates\

Temporary files: C:\Users\petr\Desktop\minifuzz\temp\

Log files: C:\Users\petr\Desktop\minifuzz\logs\

Crash files: C:\Users\petr\Desktop\minifuzz\crashes\

Aggressiveness: Low (5%)

☑ Always on Top

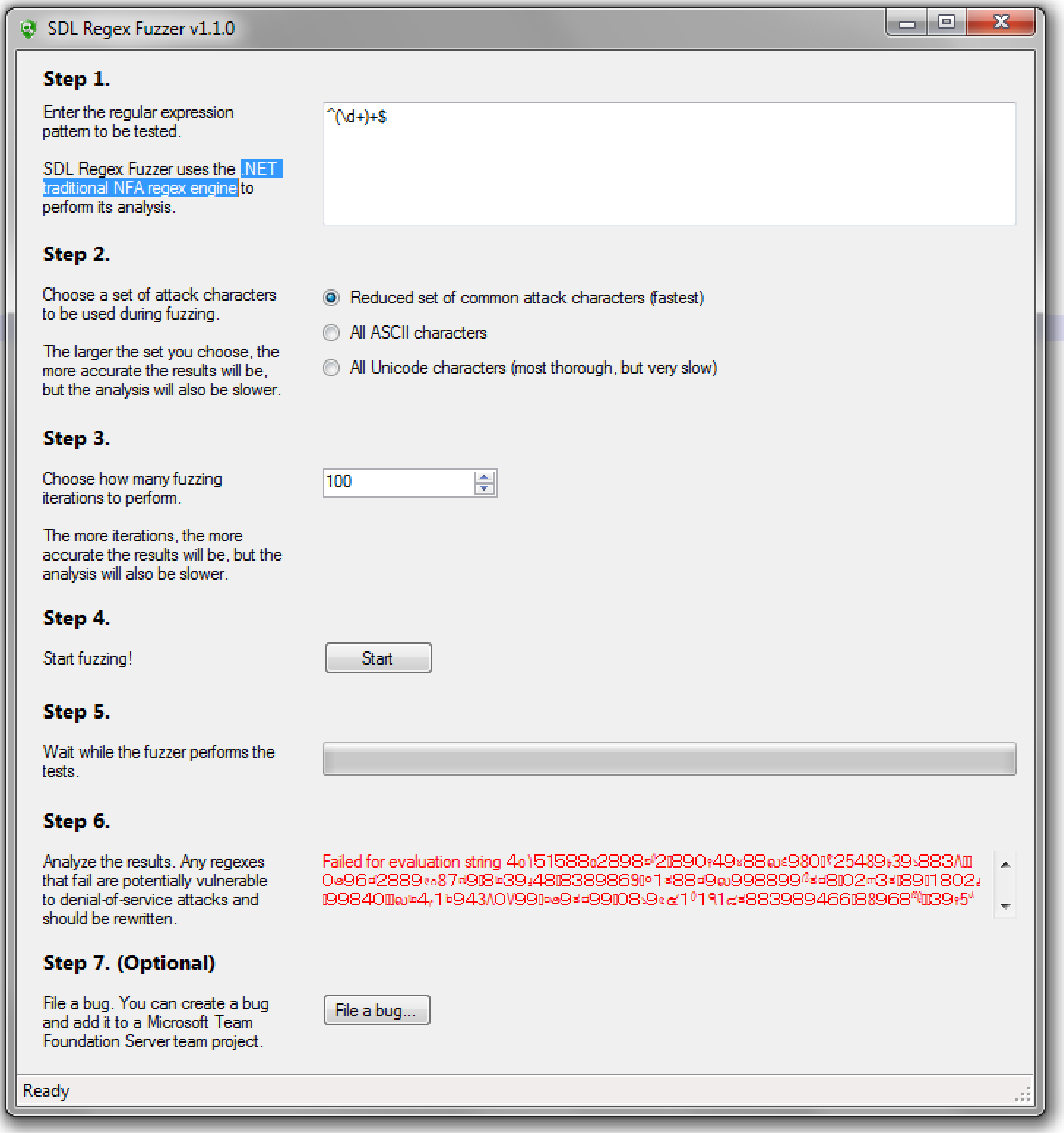
Progress

Fuzzed files: 35 # Failures: 0 main.c

Time	File	Crash

Microsoft’s SDL Regex Fuz

- Test of regular expressions evaluations
- May cause denial-of-service attack
- Use when your program use regex evaluation
- Video overview
  - http://msdn.microsoft.com/en-us/security/gg675012.aspx
- http://blogs.msdn.com/b/sdl/archive/2010/10/12/new-tool-sdl-regex-fuzzer.aspx
- Example: ^(\d+)+$



SDL Regex Fuzzer v1.1.0

Step 1.

Enter the regular expression pattern to be tested.

SDL Regex Fuzzer uses the .NET traditional NFA regex engine to perform its analysis.

```
^(\d+)+$
```

Step 2.

Choose a set of attack characters to be used during fuzzing.

The larger the set you choose, the more accurate the results will be, but the analysis will also be slower.

● Reduced set of common attack characters (fastest)

○ All ASCII characters

○ All Unicode characters (most thorough, but very slow)

Step 3.

Choose how many fuzzing iterations to perform.

The more iterations, the more accurate the results will be, but the analysis will also be slower.

100

Step 4.

Start fuzzing!

Step 5.

Wait while the fuzzer performs the tests.

Step 6.

Analyze the results. Any regexes that fail are potentially vulnerable to denial-of-service attacks and should be rewritten.

```
Failed for evaluation string 4₀15158802898²21890₁49₈88ω≤9801°25489₁39₈883/\□
0ω96²2889₈₇87⁼9□8⁼39₄48□8389869□°1⁼88⁼9ω998899⁰⁼⁼8□02⁼3⁼□89□1802₄
□99840□ω⁼4,1⁼943/\0V99□⁼ω9⁼⁼99□08₈9₄ℯ⁄1⁰1¶1ℯ⁼8839894660□8968⁰□39₁5⁴
```

Step 7. (Optional)

File a bug. You can create a bug and add it to a Microsoft Team Foundation Server team project.

Ready

SPIKE

- Tool for fuzzing analysis of network protocols
  - http://www.immunitysec.com/resources-freesoftware.shtml
- Overview of SPIKE capabilities
  - https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt
- Fuzzing tutorial with SPIKE on Vulnserver
  - http://resources.infosecinstitute.com/intro-to-fuzzing/
  - Windows & Linux version
- Another SPIKE tutorial
  - http://pentest.cryptocity.net/fuzzing/