PA193 - Secure coding principles and practices

Dynamic analysis, fuzzing

Petr Švenda svenda@fi.muni.cz

Centre for Research on Cryptography and Security

Overview

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

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
  - calls to potentially sensitive functions
- http://www.embedded.com/design/safety-and-security/4419779

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…)

Dynamic analysis tools

- Commercial
  - HP/Fortify, IBM Purify, Veracode, Coverity, Klocwork, Parasoft... (together with static analysis)
- Free
  - GCC gcov tool
  - Valgrind – set of dynamic analysis features
  - Fuzzers
- Most performance analyzers are dynamic analyzers
  - MS Visual Studio→Analyze→Start performance analysis
  - gcc -Wall -fprofile-arcs -ftest-coverage main.c
- List of tools for dynamic analysis
  - https://en.wikipedia.org/wiki/Dynamic_program_analysis

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 (later presentation)
- 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

```cpp
#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;
}
```

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

```cpp
#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;
}
```

```cpp
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 /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 2012 & GCC – runtime checks

- Corruption (usually) causes runtime exceptions
  - Stack around variable ‘Stack’ was corrupted
  - Stack around variable ‘arrayStack’ was corrupted
- MSVC: /RTC, /GS, /DYNAMICBASE (ASLR) and /NXCOMPAT (DEP)
- GCC: -fstack-protector-all, --no_execstack (DEP), kernel.randomize_va_space=1 (ASLR)

- May preventing successful exploit, but is only last defense

Cppcheck --enable=all static.cpp

```
[static.cpp:7]: (error) Array 'Static[5]' accessed at index 100, which is out of  bounds.
[static.cpp:8]: (error) Array 'Stack[5]' accessed at index 100, which is out of bounds.
[static.cpp:10]: (error) Buffer is accessed out of bounds: Stack
[static.cpp:30] -> [static.cpp:15]: (error) Array 'arrayStack[5]' accessed at
   index 100, which is out of bounds.
[static.cpp:13]: (error) Array 'array[5]' accessed at index 100, which is out of bounds.
[static.cpp:25]: (error) Memory leak: array
[static.cpp:31]: (error) Memory leak: arrayHeap
```

- (Some memory leaks also detected)

Cppcheck --enable=all file.cpp

```
#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 lea
  return 0;
}
```

```
/* Not all memory leaks are caught! */
if (1 == 2) delete[] array; /* caught */
if (Stack[0] == 1) delete[] array; /* missed */
if (Stack[0] == 1) delete[] arrayHeap; /*-//-*/
```

Visual Studio 2012 & PREfast

- Additional two problems detected
  - `Static[100] = 0;`
  - `for (int i = 0; i <= 5; i++) Stack [i] = 0;`

```
test.cpp(55): warning : C6200: Index '100' is out of valid index
   range '0' to '4' for non-stack buffer 'int * Static'.
test.cpp(58): warning : C6201: Index '5' is out of valid index
   range '0' to '4' for possibly stack allocated buffer 'Stack'.
test.cpp(55): warning : C6386: Buffer overrun while writing to 'Static':
   the writable size is '20' bytes, but '404' bytes might be written.
test.cpp(62): warning : C6386: Buffer overrun while writing to 'array':
   the writable size is '5*4' bytes, but '404' bytes might be written.
```

- arrayStack and arrayHeap overruns still missed

Visual Studio 2012 & PREfast

```
#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;
}
```

Visual Studio 2012 & 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 2012 & 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*, unsigned int) (test.cpp:17)
==    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_malloc.c:363)
==    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 size 4
==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 after Expected
==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

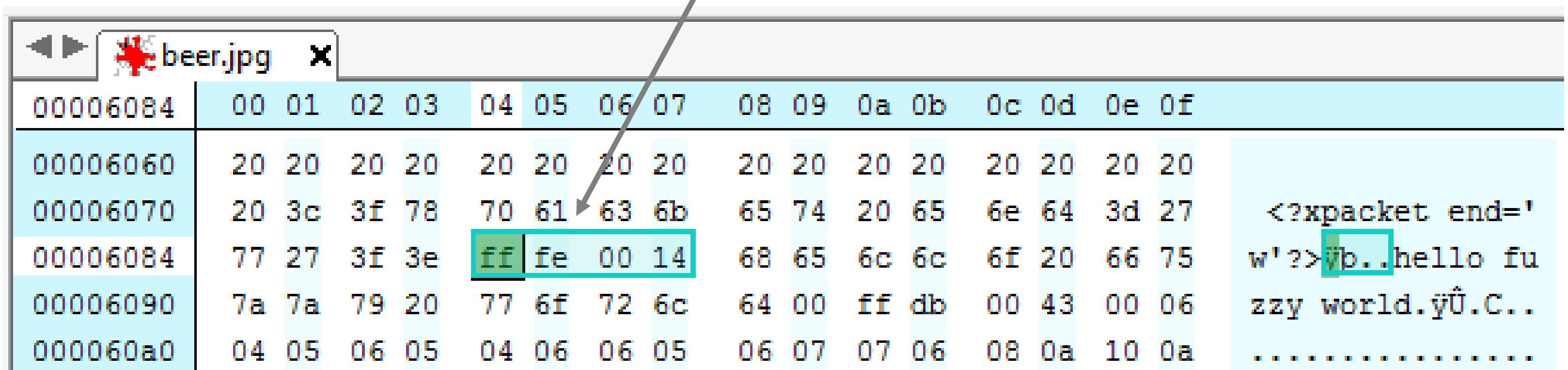
- *Compilers* (MSVC, GCC) will miss many problems
- *Compiler flags (/RTC* and */GS;* **`-fstack-protector-all`***)* flags
  - detect (some) stack based corruptions at runtime
  - additional preventive flags /DYNAMICBASE (ASLR) and /NXCOMPAT (DEP)
- *Valgrind memcheck*
  - will not find stack based problems, 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)

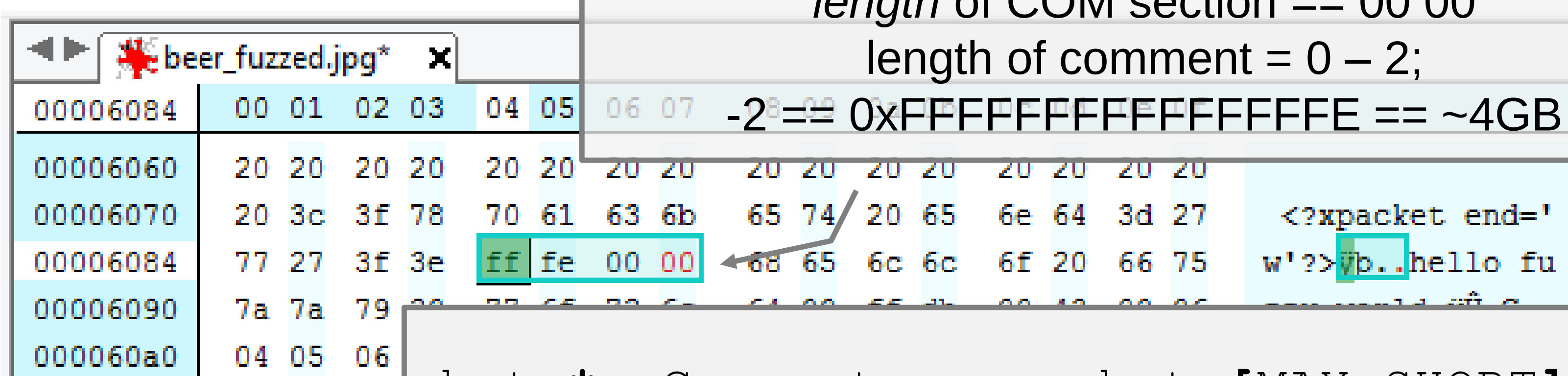

Beer
HELPING ENGLISH PEOPLE
HAVE SEX SINCE 1842!
THE ENGLISH PUB
Austurvöllur

What is wrong?

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



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 input checking
- Type signed/unsigned mismatch
- Type overflow
- Buffer overflow
- Heap overflow
- Source code was not available (blackbox testing)
- Huge impact (core MS library)
- Easily exploitable

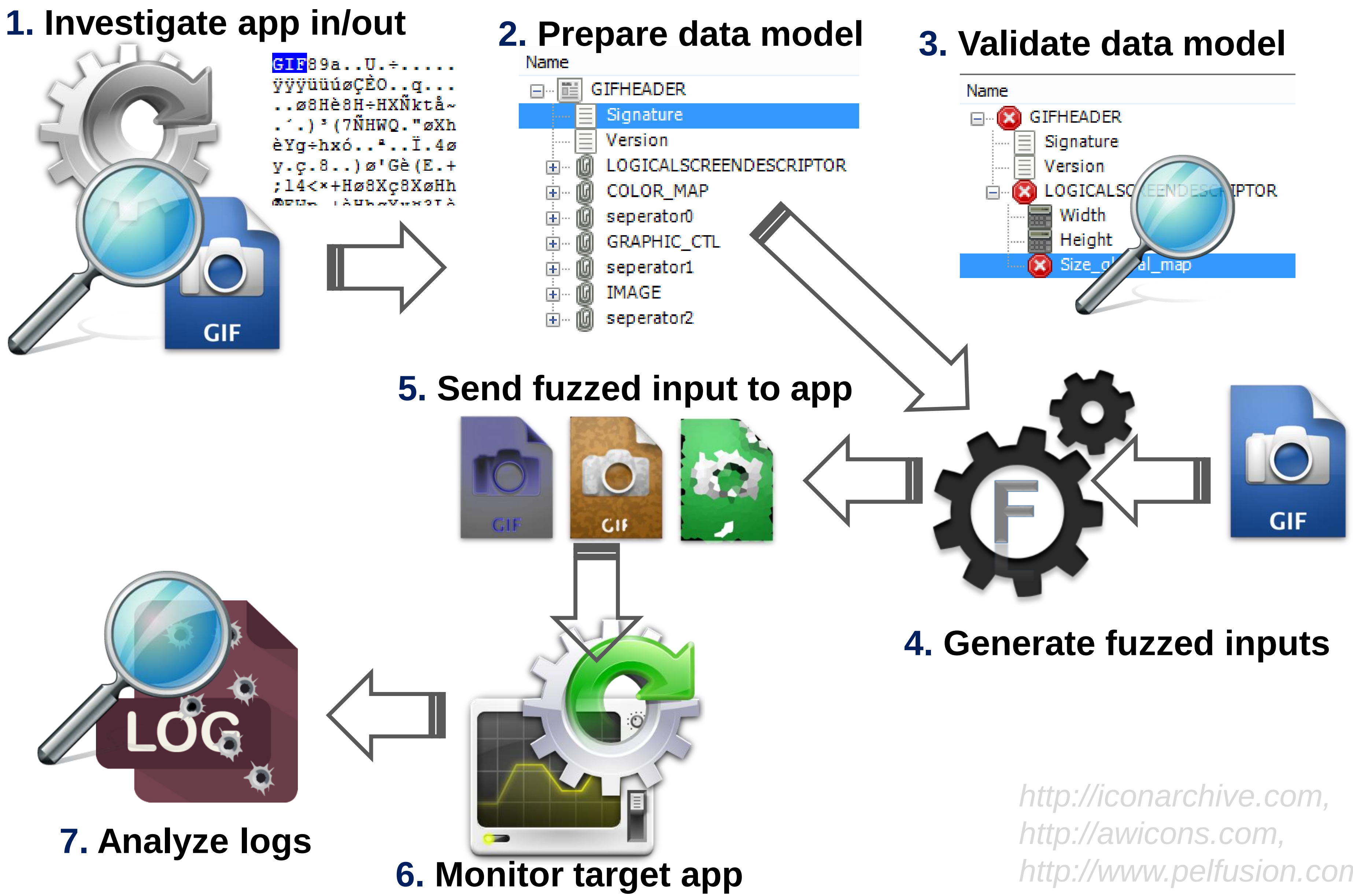
FOUND BY FUZZING ☺

INTRO TO FUZZING

Very simple fuzzer

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

What do you miss here?


1. Investigate app in/out
GIF89a..U.÷.....
2. Prepare data model
Name
GIFHEADER
Signature
Version
LOGICALSCREENDESCRIPTOR
COLOR_MAP
seperator0
GRAPHIC_CTL
seperator1
IMAGE
seperator2
3. Validate data model
Name
GIFHEADER
Signature
Version
Width
Height
GIF
5. Send fuzzed input to app
GIF
GIF
4. Generate fuzzed inputs
GIF
LOG
7. Analyze logs
6. Monitor target app
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 impl.)

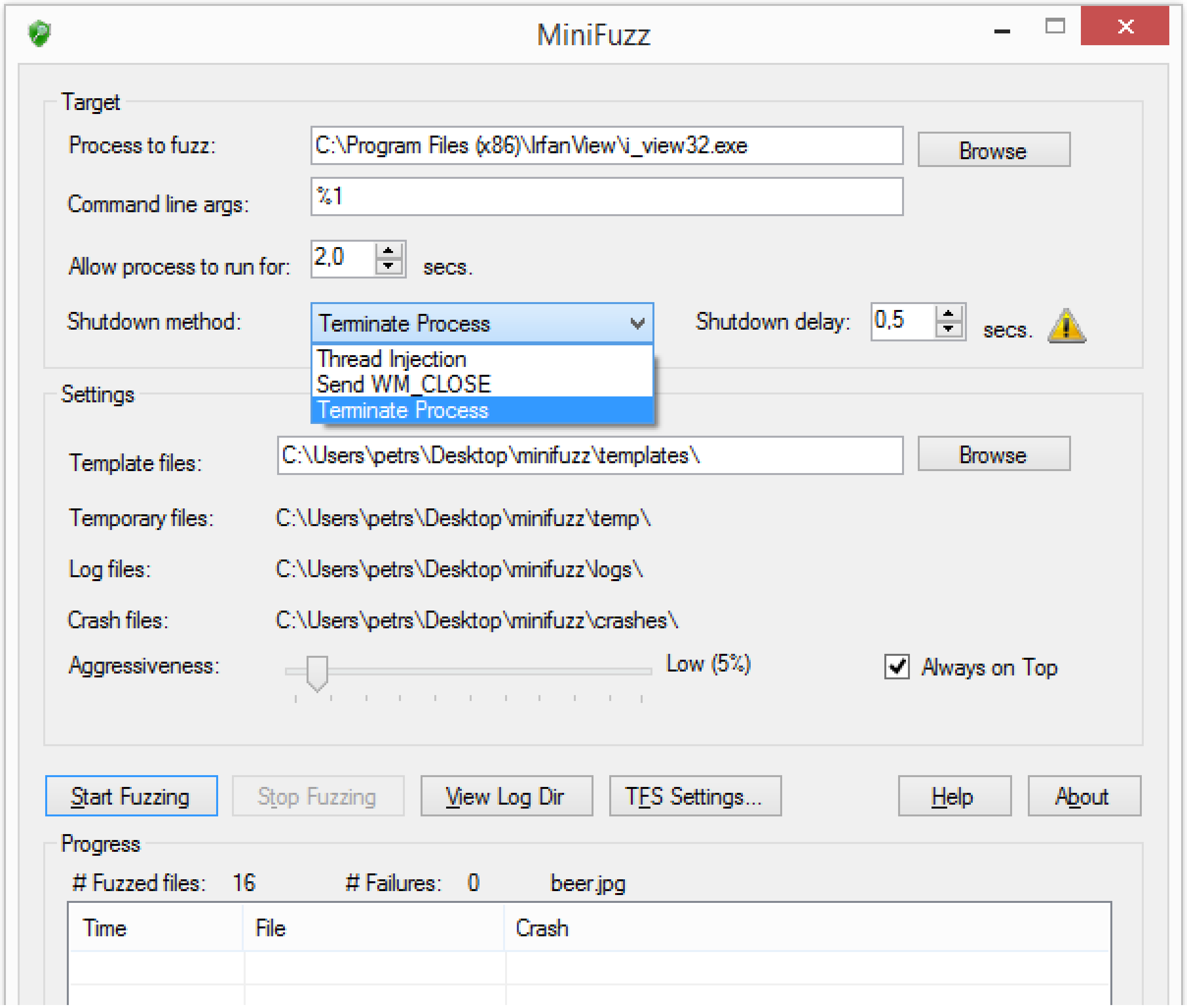
What kind of bugs are usually missed?

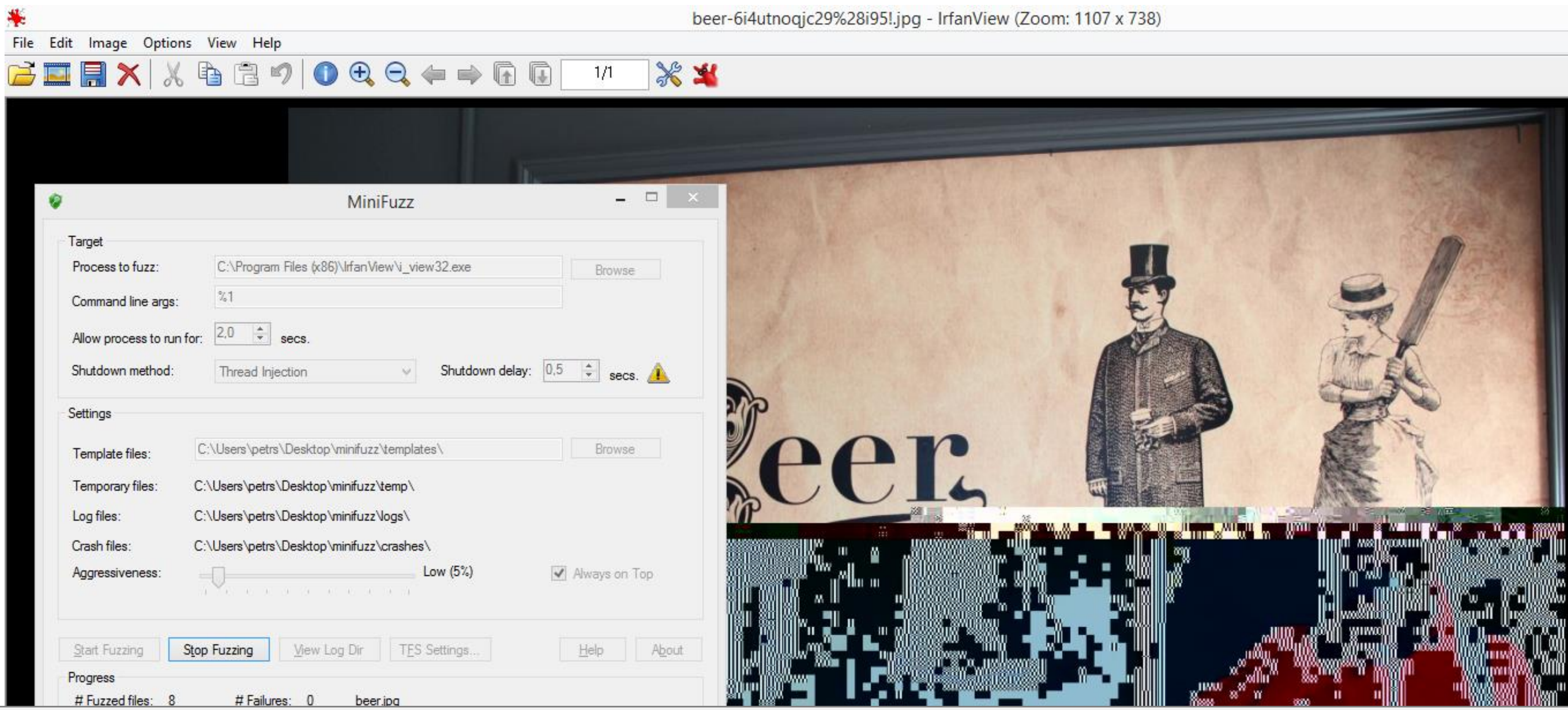
- Bugs after input validation (if not modeled properly)
- High-level / architecture bugs (e.g. weak crypto)
- Usability bugs
- …

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

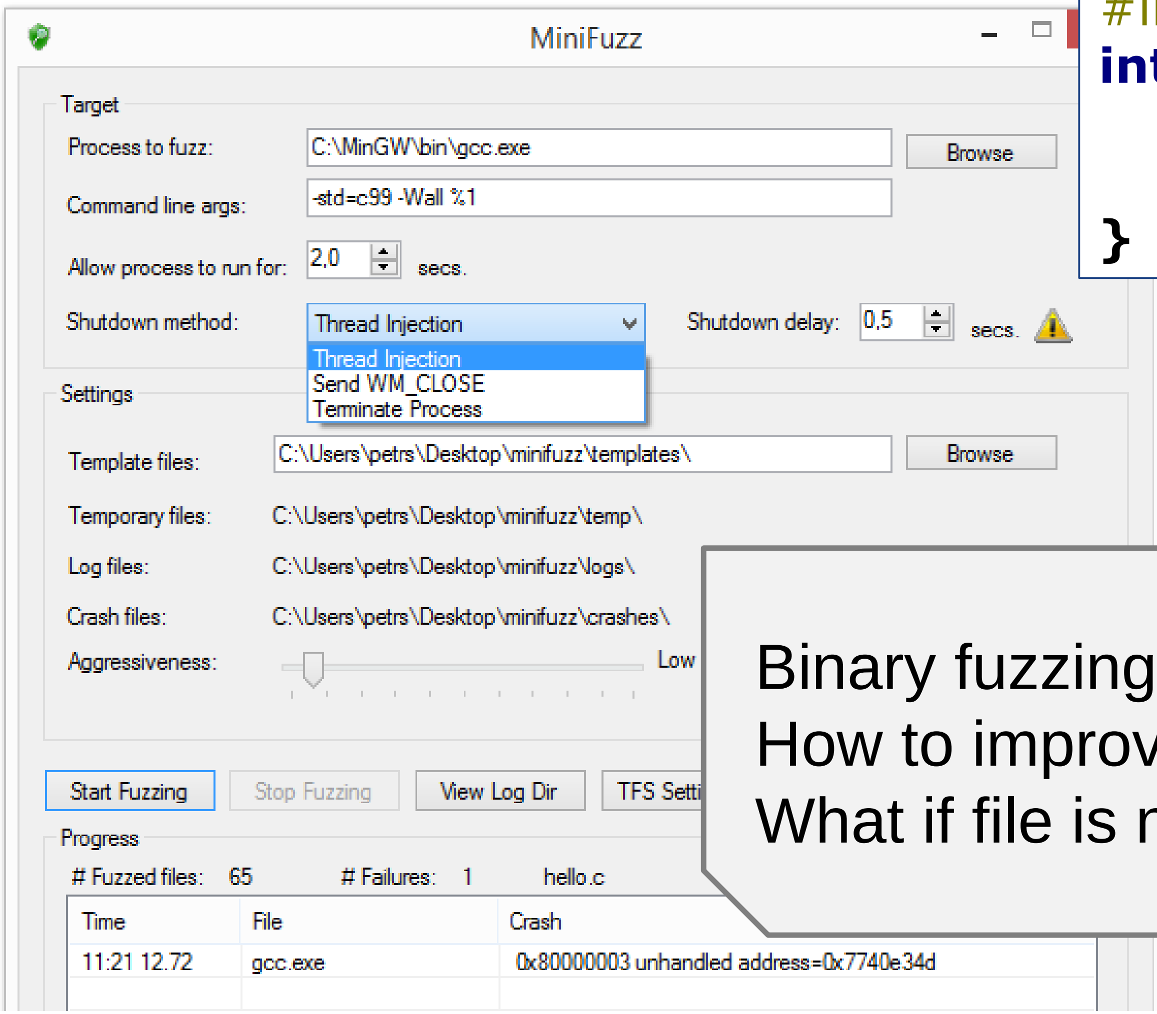
Microsoft's SDL MiniFuzz File Fuzzer





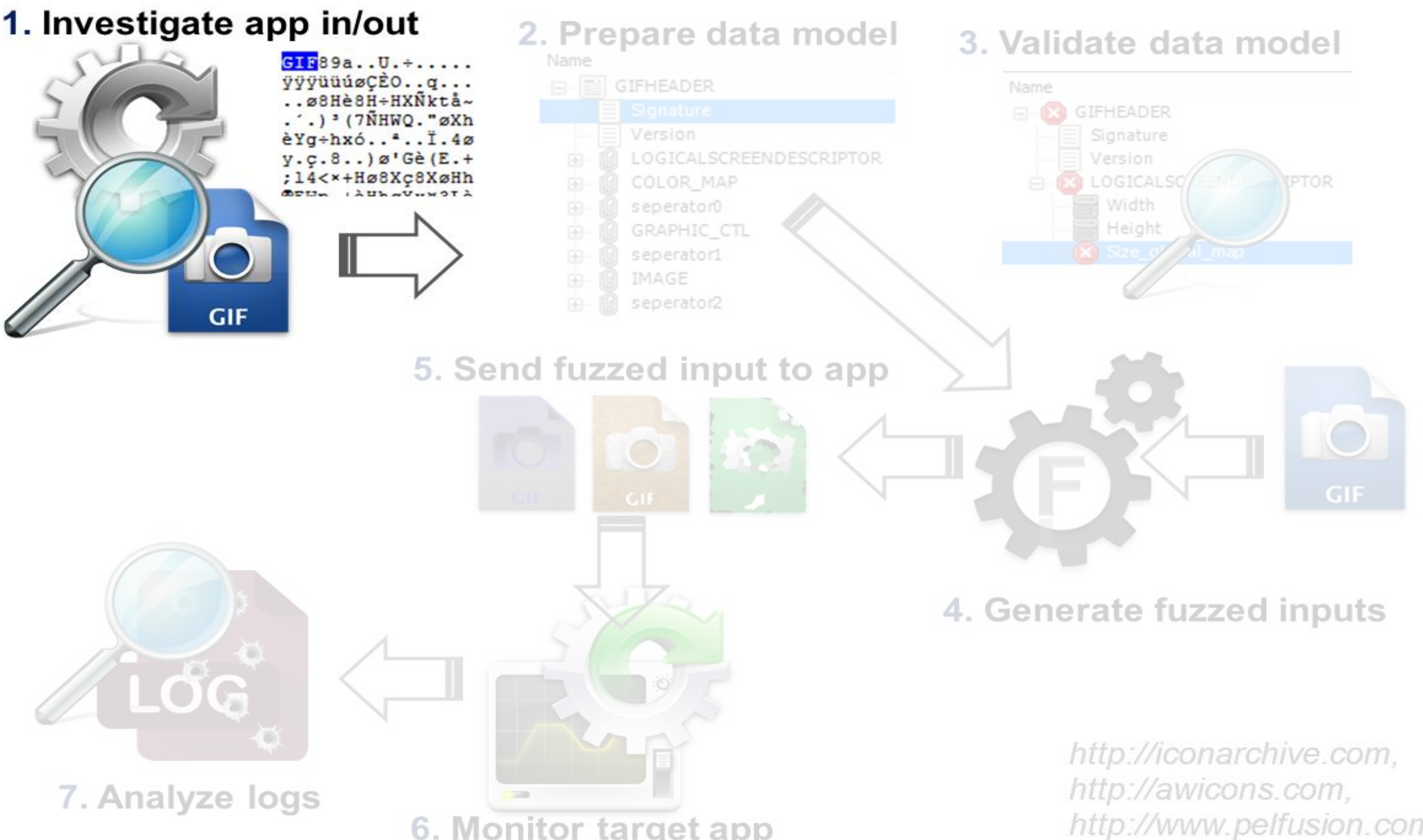

```xml
<?xml version="1.0"?>
<failures>
  <failure type="Exception Event:Tid=8504, 0x80000003, unhandled, address=0x7740e34d" datetime="11:21:12 12.
    <registers RAX="00000000" RBX="00000000" RCX="7FFF5FC5180A" RDX="00000000" RSI="00000000" RDI="000
    <process name="C:\Program Files (x86)\IrfanView\i_view32.exe"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

MiniFuzz: gcc fuzzing




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

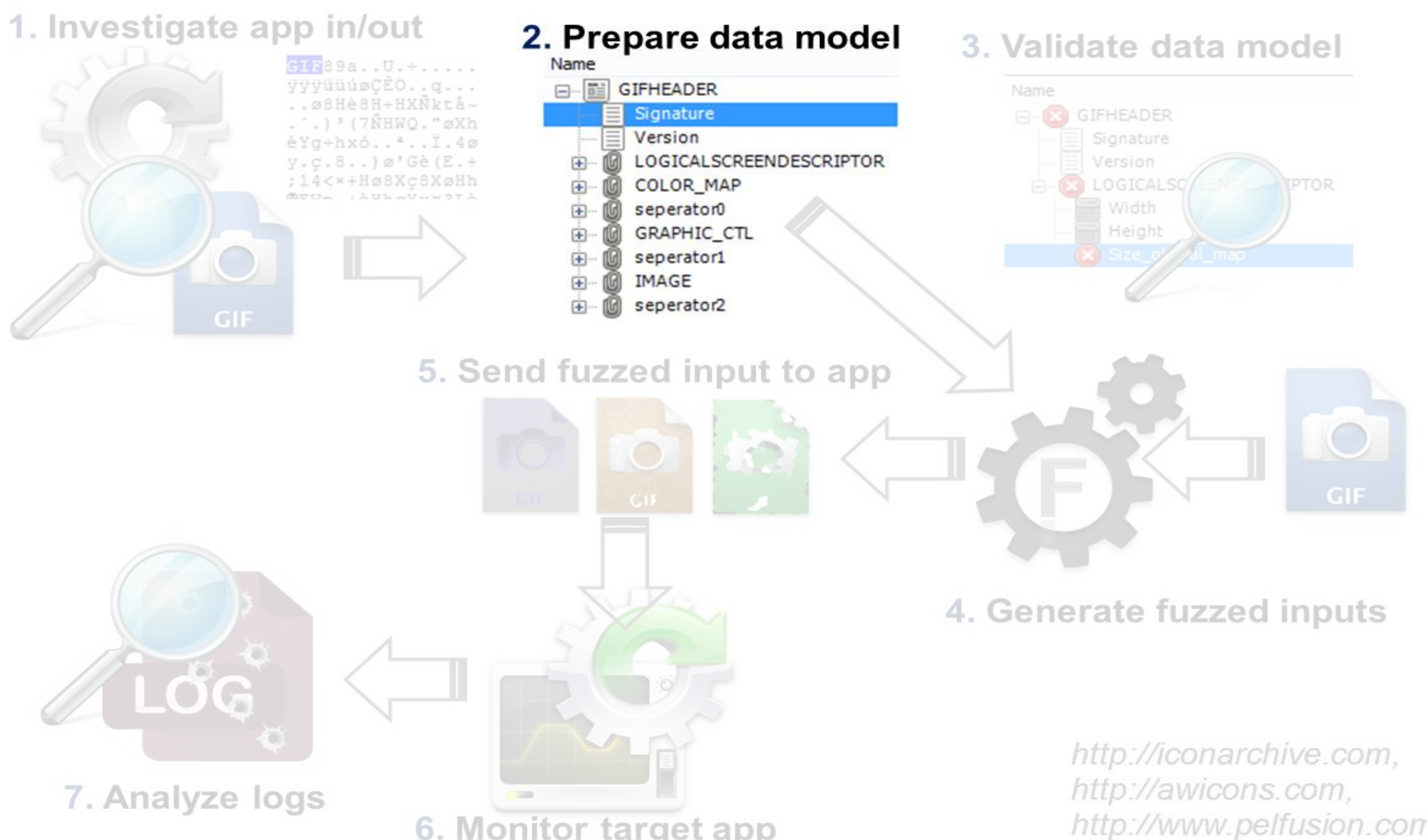
Binary fuzzing of source code???
How to improve test coverage?
What if file is not an input?



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?


1. Investigate app in/out
GIF
2. Prepare data model
Name
GIFHEADER
Signature
Version
LOGICALSCREENDESCRIPTOR
COLOR_MAP
seperator0
GRAPHIC_CTL
seperator1
IMAGE
seperator2
3. Validate data model
Name
GIFHEADER
Signature
Version
Width
Height
5. Send fuzzed input to app
GIF
GIF
4. Generate fuzzed inputs
LOG
7. Analyze logs
6. Monitor target app
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))
```

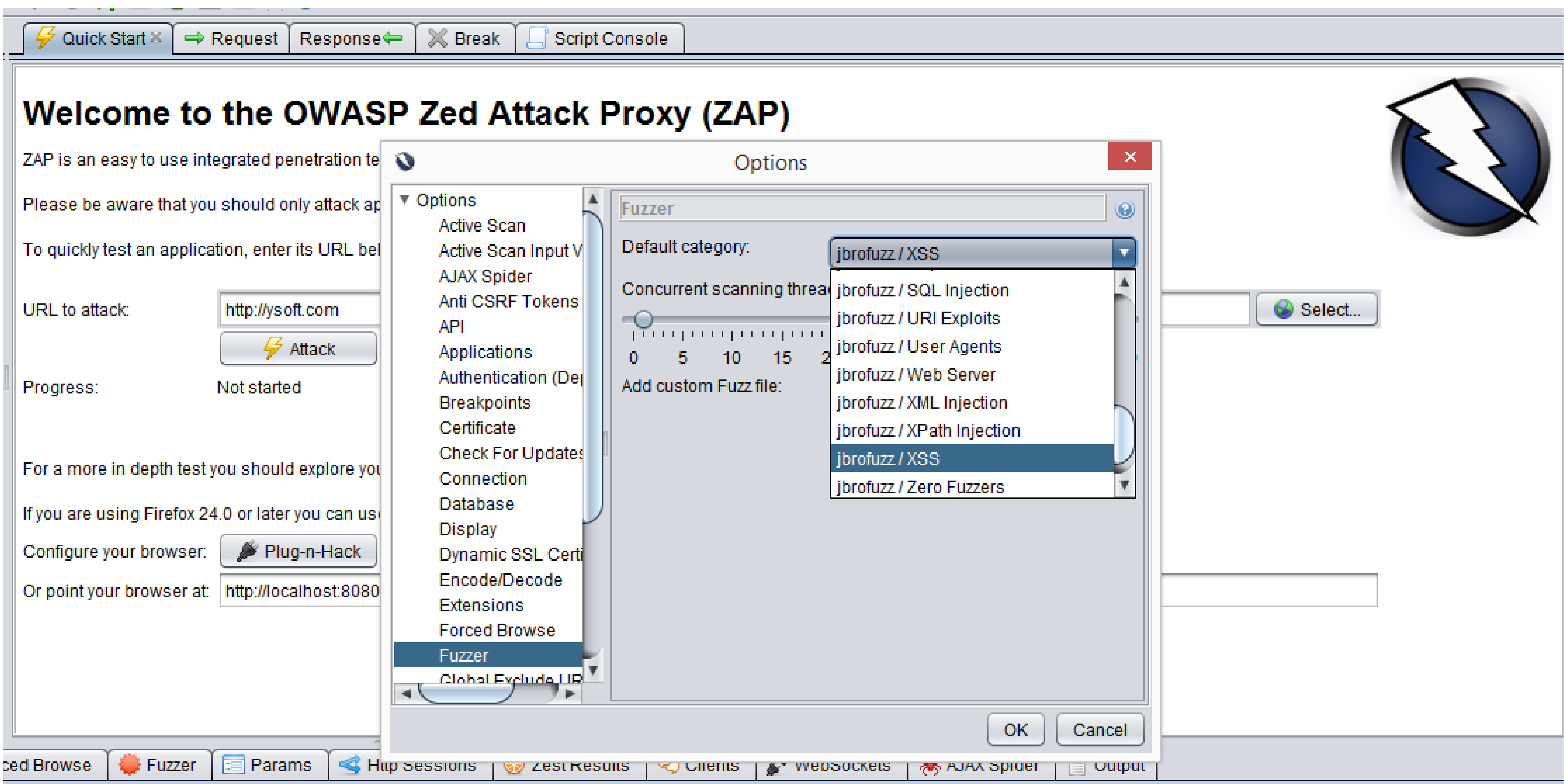
How to generate fuzzed input?

- Generational fuzzing (Recursive fuzzing)
  - Produces data based only on data model description
  - E.g., iterates over range of values of given alphabet
- Mutational fuzzing (Replacive fuzzing)
  - Produces data based on templates and supplied model
  - Known border values or malicious malformed input
  - Fuzz test vectors
  - String-based mutators, number-based mutators…

Fuzzing via intermediate proxy

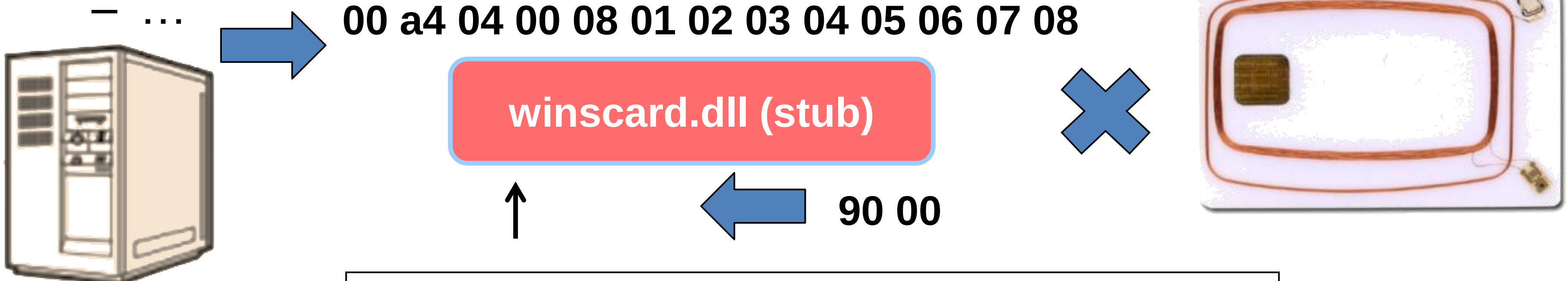
- Fuzzer modifies valid flow according to data model
- Usually used for fuzzing of state-full protocols
  - Modelling states and interactions would be difficult
  - Target application(s) takes care of states and valid input

OWASP's ZAP – fuzz strategy settings

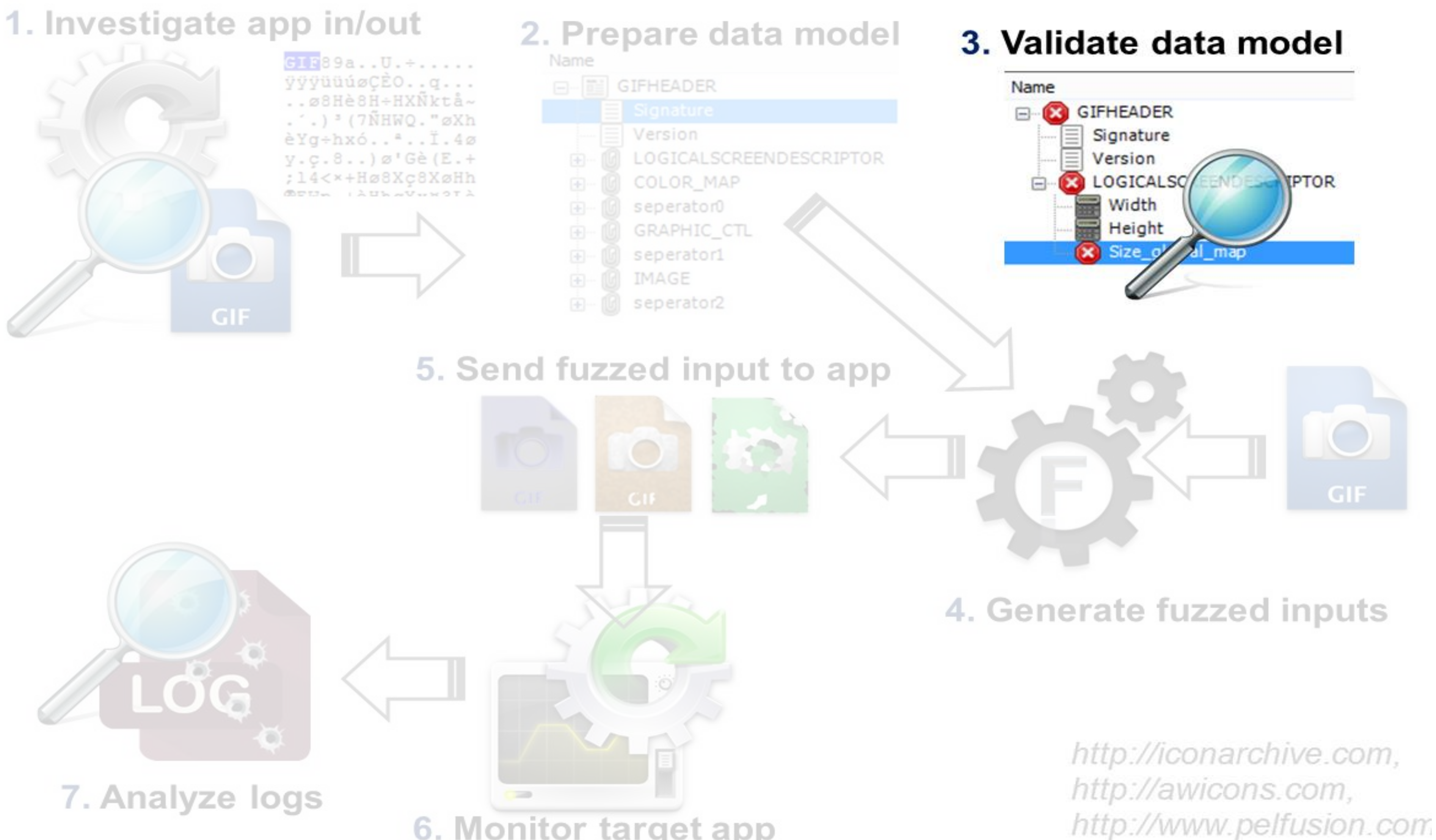


APDUPlay - Smart card fuzzing

- 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
  - …

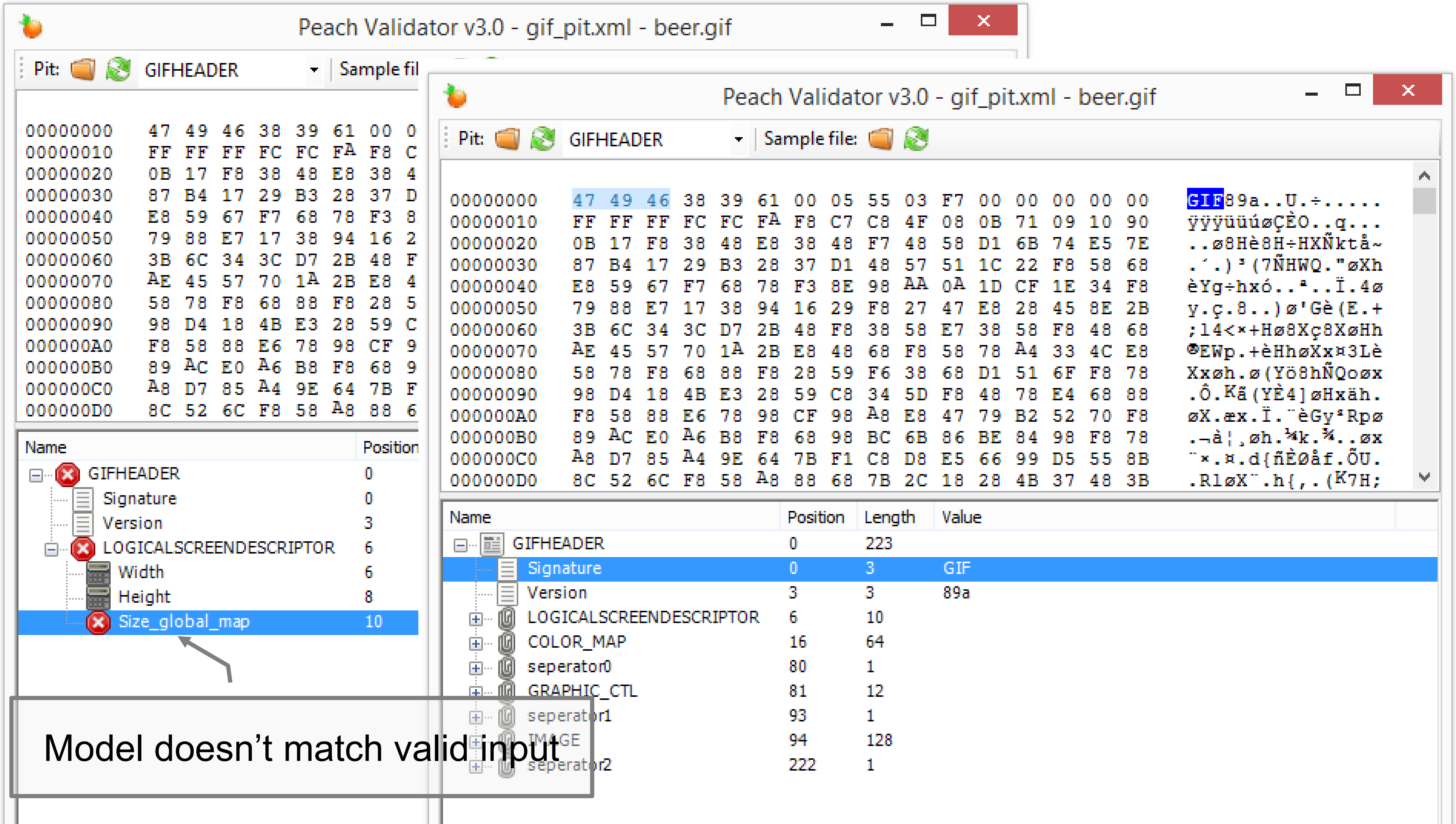



```
[RULE1]
MATCH1=in=1;t=0;cla=00;ins=a4;p1=04;
ACTION=in=0;data0=90 00;le=02;
```


1. Investigate app in/out
GIF
2. Prepare data model
Name
GIFHEADER
Signature
Version
LOGICALSCREENDESCRIPTOR
COLOR_MAP
seperator0
GRAPHIC_CTL
seperator1
IMAGE
seperator2
3. Validate data model
Name
GIFHEADER
Signature
Version
Width
Height
4. Generate fuzzed inputs
GIF
5. Send fuzzed input to app
6. Monitor target app
LOG
7. Analyze logs
http://iconarchive.com,
http://awicons.com,

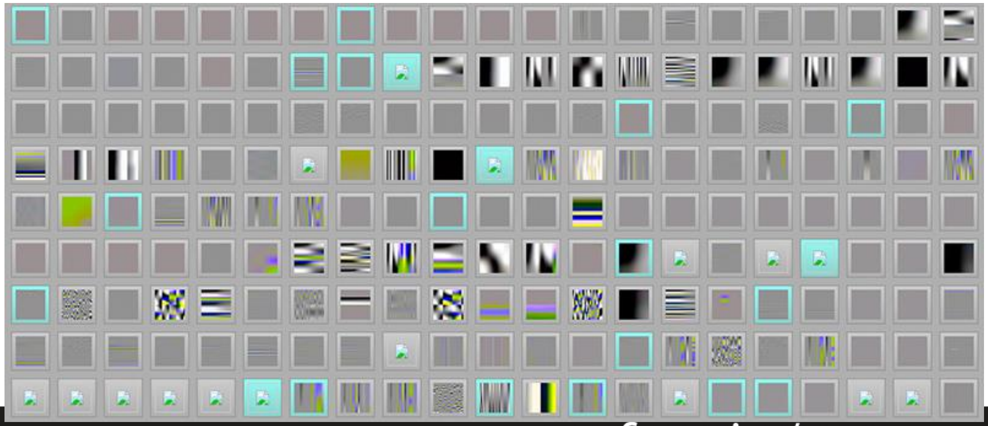
VALIDATION

Peach Validator 3.0



American fuzzy lop

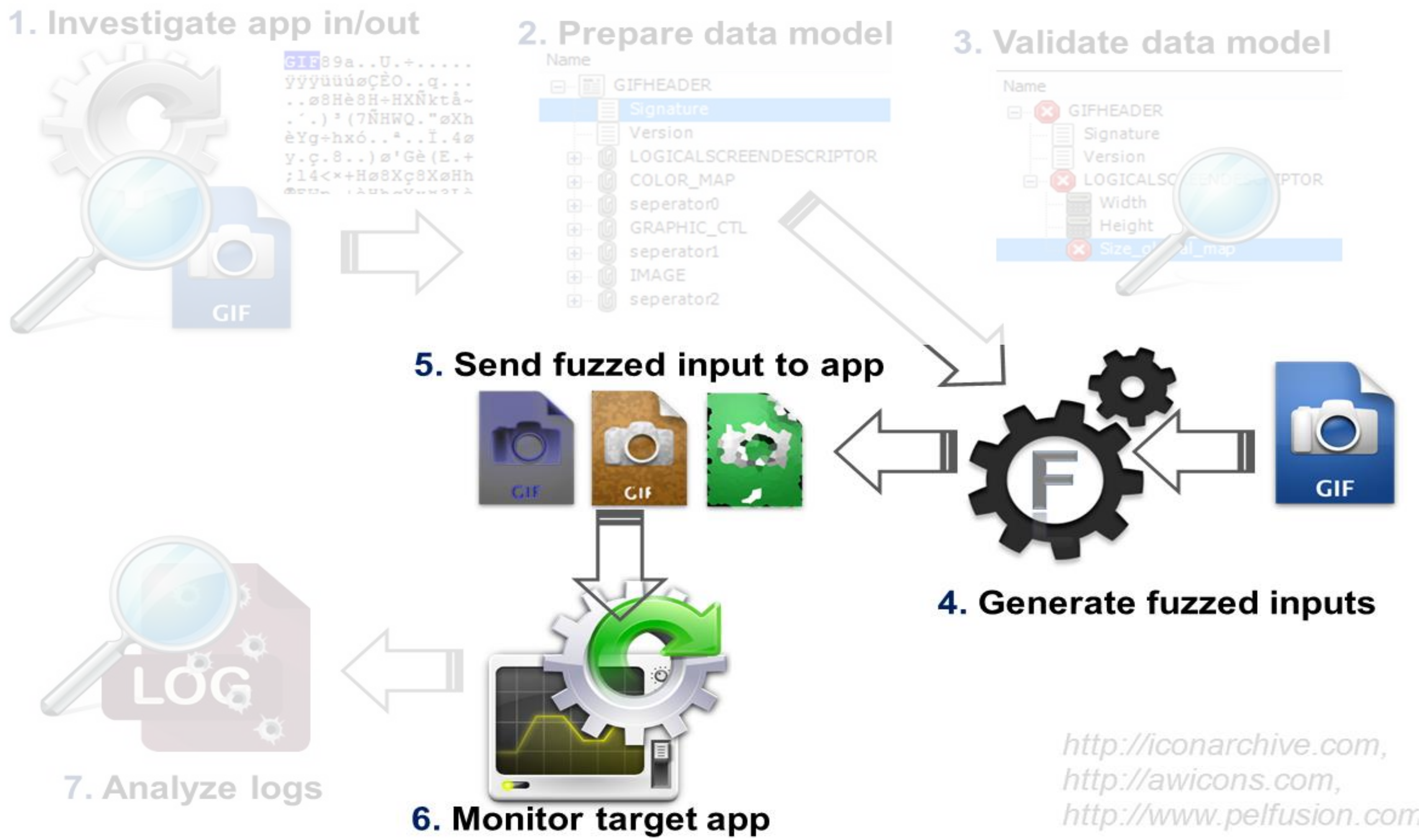
- New, but actively developed tool
- High speed fuzzer http://lcamtuf.coredump.cx/afl/
- Sophisticated generation of test cases (coverage)
- Automatic generation of input templates
  - E.g., valid JPEG image from “hello” string after few days
  - http://lcamtuf.blogspot.cz/2014/11/pulling-jpegs-out-of-thin-air.html
- 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



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

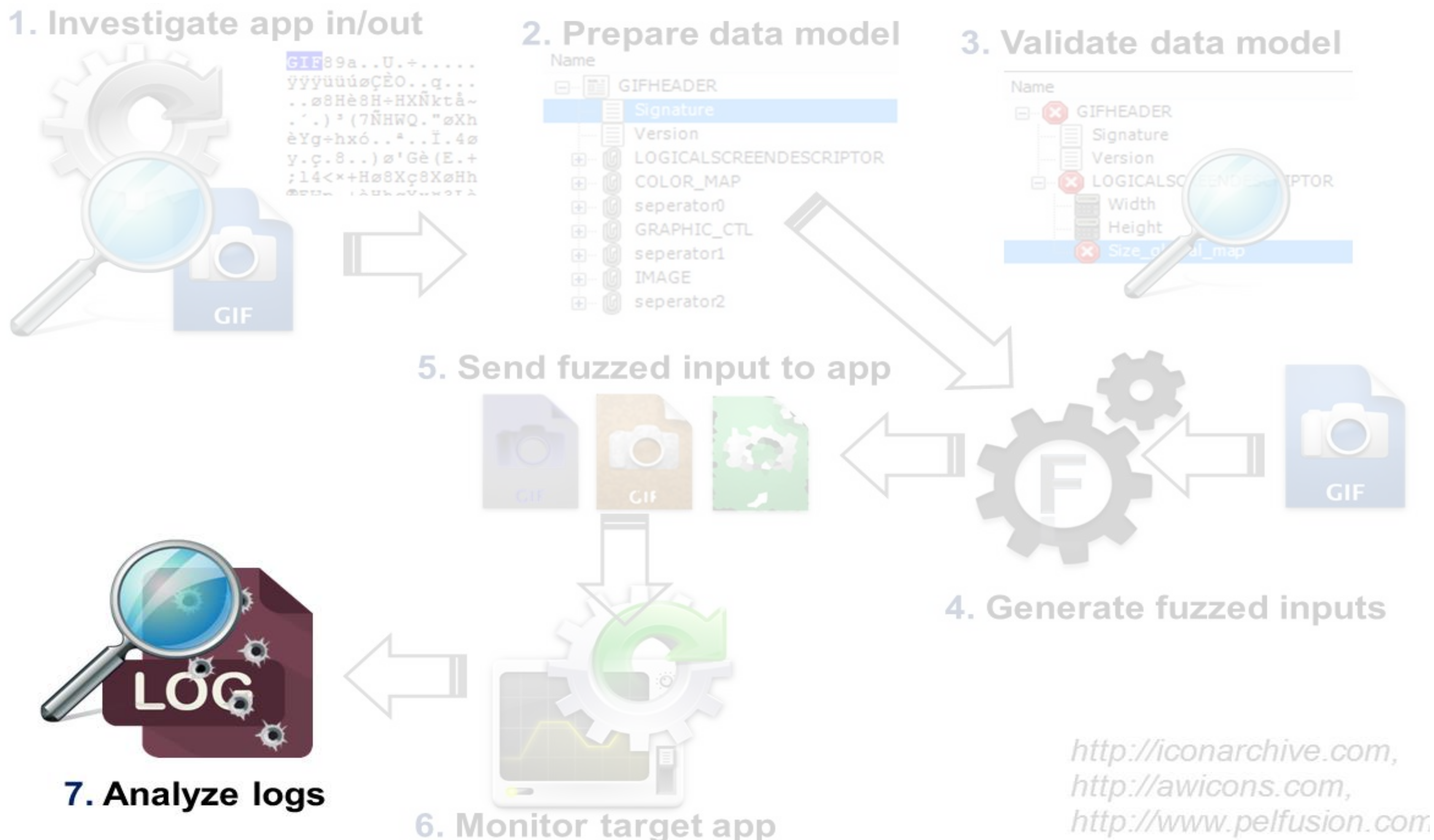
- OS X Crash Wrangler Monitor
- OS X Crash Reporter Monitor

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


1. Investigate app in/out
GIF
2. Prepare data model
Name
GIFHEADER
Signature
Version
LOGICALSCREENDESCRIPTOR
COLOR_MAP
seperator0
GRAPHIC_CTL
seperator1
IMAGE
seperator2
3. Validate data model
Name
GIFHEADER
Signature
Version
Width
Height
5. Send fuzzed input to app
GIF
4. Generate fuzzed inputs
LOG
7. Analyze logs
6. Monitor target app
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

- 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 steaper learning curve
- If fuzzing will not find any bugs, check your model
- Try it!

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 ☹

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"

Execution of sensitive operation

- Before sensitive operation (e.g., system()) is executed with value, taint label is checked
  - if value is tainted, alert is issued
- Untrusted data reaching privilege location is detected
  - can detect even unknown attacks
  - (but sometimes we need to use user input)

Taint analysis - tools

- Taintgrind
  - http://www.cl.cam.ac.uk/~wmk26/taintgrind/
  - additional module to Valgrind
  - dynamic taint analyzer for C/C++
  - output memory traces (information flows) already produced by Valgrind
- Tanalysis
  - http://code.google.com/p/tanalysis/
  - static taint analyzer for C
  - plugin for Frama-C http://frama-c.com/
- Read more about taint analysis
  - http://users.ece.cmu.edu/~ejschwar/papers/oakland10.pdf

Microsoft PREfast + Taint analysis

- Warning C6029 is issued when tainted value is passed to parameter marked as [Post(Tainted=No)]
  - without any checking (any condition statement)
- http://msdn.microsoft.com/en-us/library/ms182047%28v=vs.100%29.aspx

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(Tainted=SA_Yes)] int c);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(Tainted=Yes)] int c);
```

Coverity taint analysis

- TAINTED_SCALAR
  - http://blog.coverity.com/2014/04/18/coverity-heartbleed-part-2/#.U1l4k2dOURo
  - http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html

Conclusions

- Dynamic analyzers can profile application
  - and find bugs not found by static analysis
- Fuzzing is “cheap” blackbox approach via malformed inputs

Questions

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=Bm3Mfndrl1Y
- 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

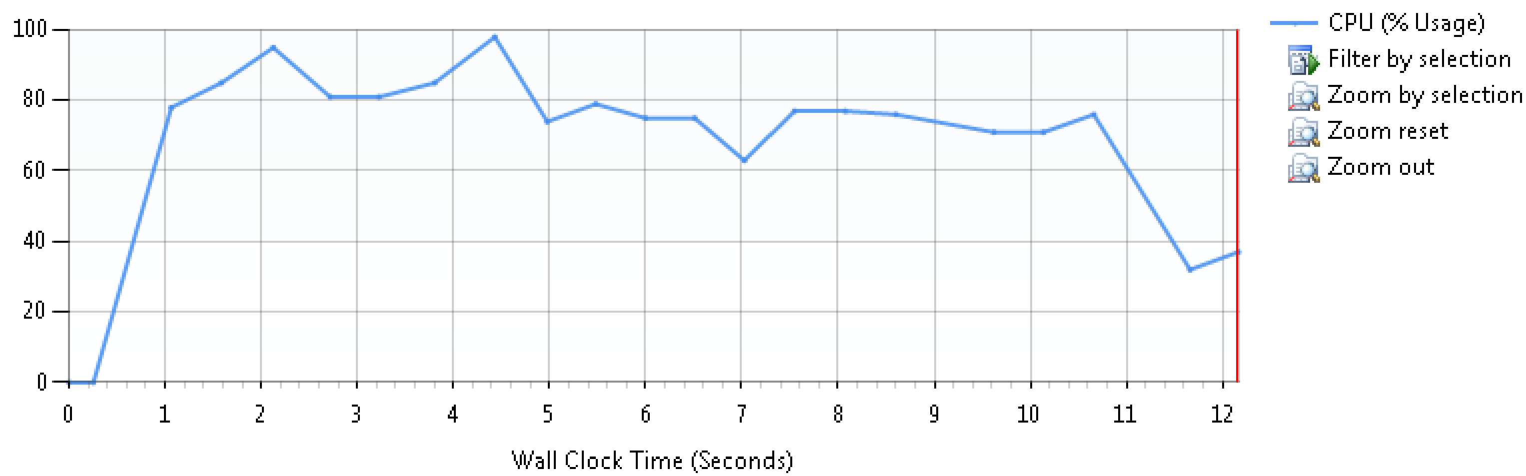
- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
  - what function called actual one
  - based on real code execution (condition jumps)
- Many other statistics, depend on the tools
- Helps to focus and scope security analysis

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)

CROCS

MS VS Profiler – results (Functions)

pb173_aes101115.vsp | time.h | aes32.h | pb173_aes.cpp

Current View: 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)	1,668	14	95.86	0.80
rj_sbox(unsigned char)	1,694	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

		Hit	Total	Coverage
Current view: top level - example/methods - iterate.c (source / functions)				
Test: Basic example (view descriptions)	**Lines:**	8	8	100.0 %
Date: 2012-10-12	**Functions:**	1	1	100.0 %
Legend: Lines: hit not hit | Branches: + taken - not taken # not executed	**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