PV286 - Secure coding principles and practices

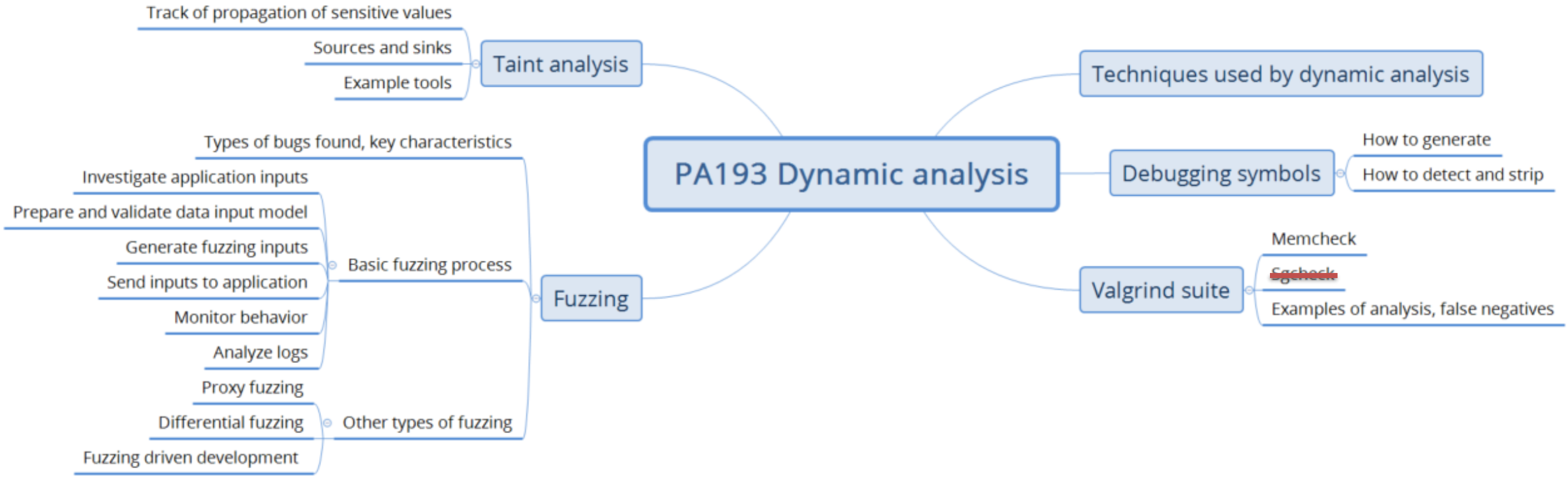
Dynamic analysis, fuzzing, and taint analysis

Łukasz Chmielewski *chmiel@fi.muni.cz* *(based on the lecture by P. Svenda)*
(email me with your questions/feedback)
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on Cryptography and Security

This Lecture

- Today we cover dynamic analysis of source code
- First Jan Kvapil will give a presentation about the project.
  - Milan Šorf, Roman Lacko, Štěpánka Trnková, Jiří Gavenda, Tomáš Jaroš, and Antonín Dufka.
- Resources:
  - I will attempt to record the lecture and if it works it should be available around Wednesday.
  - An older (but well-recorded) version of the lecture from 2022 (by P. Švenda):
    - https://is.muni.cz/auth/player?lang=en;furl=%2Fel%2Ffi%2Fjaro2022%2FPA193%2Fum%2Fvideo%2FPA193_02_DynamicAnalysisFuzzing_2022.video5
  - Last year (worse quality):
    - https://is.muni.cz/auth/el/fi/jaro2023/PV286/um/vi/137055932/
  - Materials:
    - https://is.muni.cz/auth/el/fi/jaro2024/PV286/um/


PA193 Dynamic analysis
Techniques used by dynamic analysis
Debugging symbols
How to generate
How to detect and strip
Valgrind suite
Memcheck
~~Sgcheck~~
Examples of analysis, false negatives
Fuzzing
Types of bugs found, key characteristics
Basic fuzzing process
Investigate application inputs
Prepare and validate data input model
Generate fuzzing inputs
Send inputs to application
Monitor behavior
Analyze logs
Other types of fuzzing
Proxy fuzzing
Differential fuzzing
Fuzzing driven development
Taint analysis
Track of propagation of sensitive values
Sources and sinks
Example tools

DYNAMIC ANALYSIS

Static vs. dynamic analysis

- Static analysis
  - examine program's code without executing it
  - can examine both source code and compiled code
    - source code is easier to understand (more metadata)
  - can be applied on unfinished code
  - manual code audit is kind of static analysis
- Dynamic analysis
  - code is executed (compiled or interpreted)
  - input values are supplied, internal memory is examined…

What can dynamic analysis provide

- Dynamic analysis compiles and executes 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
  - MSVC: Do not provide .pdb files
  - GCC: check compiler flags, use **`strip`** command
- Check for debugging symbols
  - 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 call 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, 5);
  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 (later versions) /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 & 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 & 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
==17239== Invalid write of size 4
==17239==    at 0x4006AB: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:14)
==17239==    by 0x40075D: main (test.cpp:33)
==17239==  Address 0x595f230 is not stack'd, malloc'd or (recently) free'd
==17239==
==17239== Invalid write of size 4
==17239==    at 0x4006CB: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:17)
==17239==    by 0x40075D: main (test.cpp:33)
==17239==  Address 0x595f1d0 is not stack'd, malloc'd or (recently) free'd
==17239==
==17239== Invalid write of size 4
==17239==    at 0x400710: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:23)
==17239==    by 0x40075D: main (test.cpp:33)
==17239==  Address 0x595f054 is 0 bytes after a block of size 20 alloc'd
==17239==    at 0x4C28152: operator new[](unsigned long) (vg_replace_malloc.c:395)
==17239==    by 0x40073F: main (test.cpp:32)
...
==17239== LEAK SUMMARY:
==17239==    definitely lost: 40 bytes in 2 blocks
...
==17239== 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;
}
```

Sgcheck removed from Valgrind Release 3.16.0 (27 May 2020)

- https://www.valgrind.org/docs/manual/dist.news.html
- “The experimental Stack and Global Array Checking tool has been re moved. It only ever worked on x86 and amd64, and even on those it had a high false positive rate and was slow.”

- Takeaway: Some methods will be too costly or with too much overhead or with too many false positives (problem to solve is hard)

(MSVS) _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.
```

https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/crtdumpmemoryleaks?view=msvc-170

Dr.Memory memory analysis (https://drmemory.org/)

- Can run as standalone tool or Visual Studio plugin
- Targets primarily C and C++ binaries
- Also capable of fuzzing
  - Selected separate function from target binary, define fuzzing methodology
  - https://drmemory.org/page_fuzzer.html

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* (removed 27.5.2020)
  - 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”) == ?

beer.jpg

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	....

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

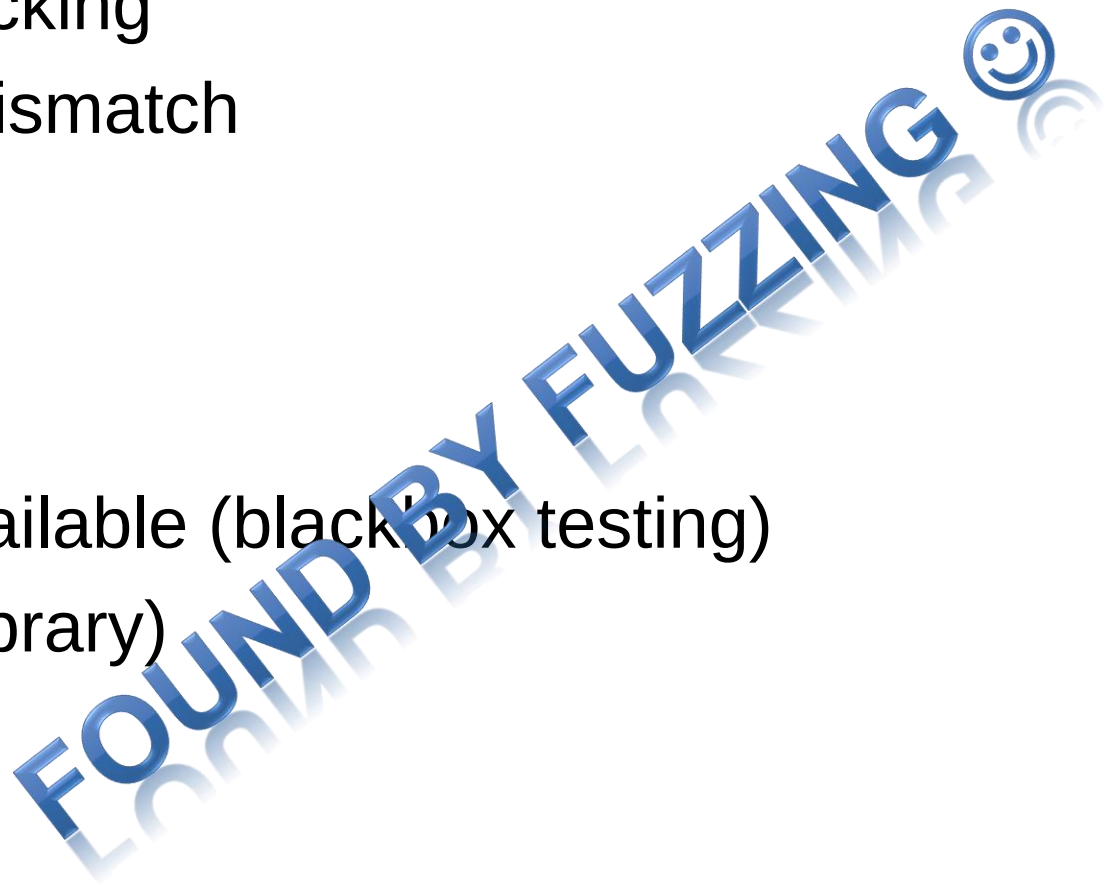
beer_fuzzed.jpg*

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	00	68	65	6c	6c	6f	20	66	75	w'?>ÿþ..hello fu
00006090	7a	7a	79														
000060a0	04	05	06														

```
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 (black box testing)
- Huge impact (core MS library)
- Easily exploitable



INTRO TO FUZZING

Very simple fuzzer

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

What do you miss here?

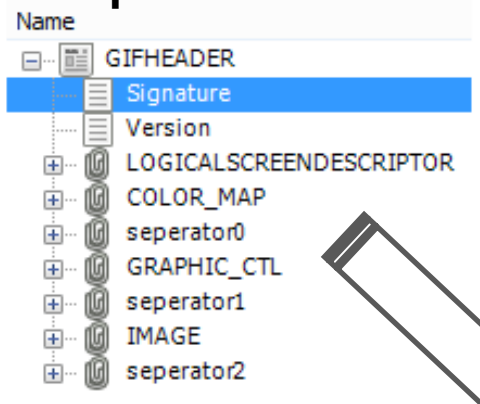
What is missing?

- 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)

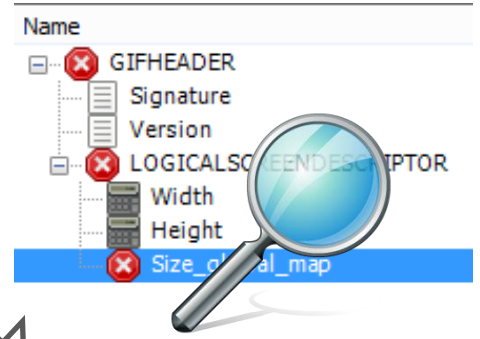
1. Investigate app in/out

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

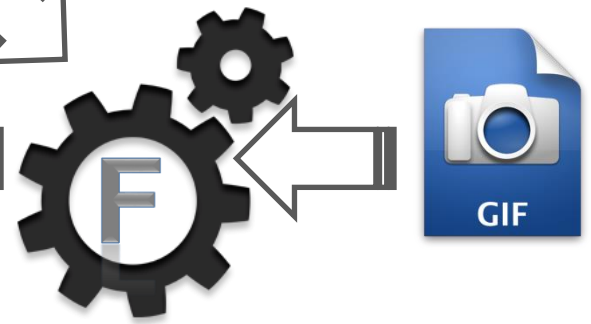
2. Prepare data model (optional)



3. Validate data model



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

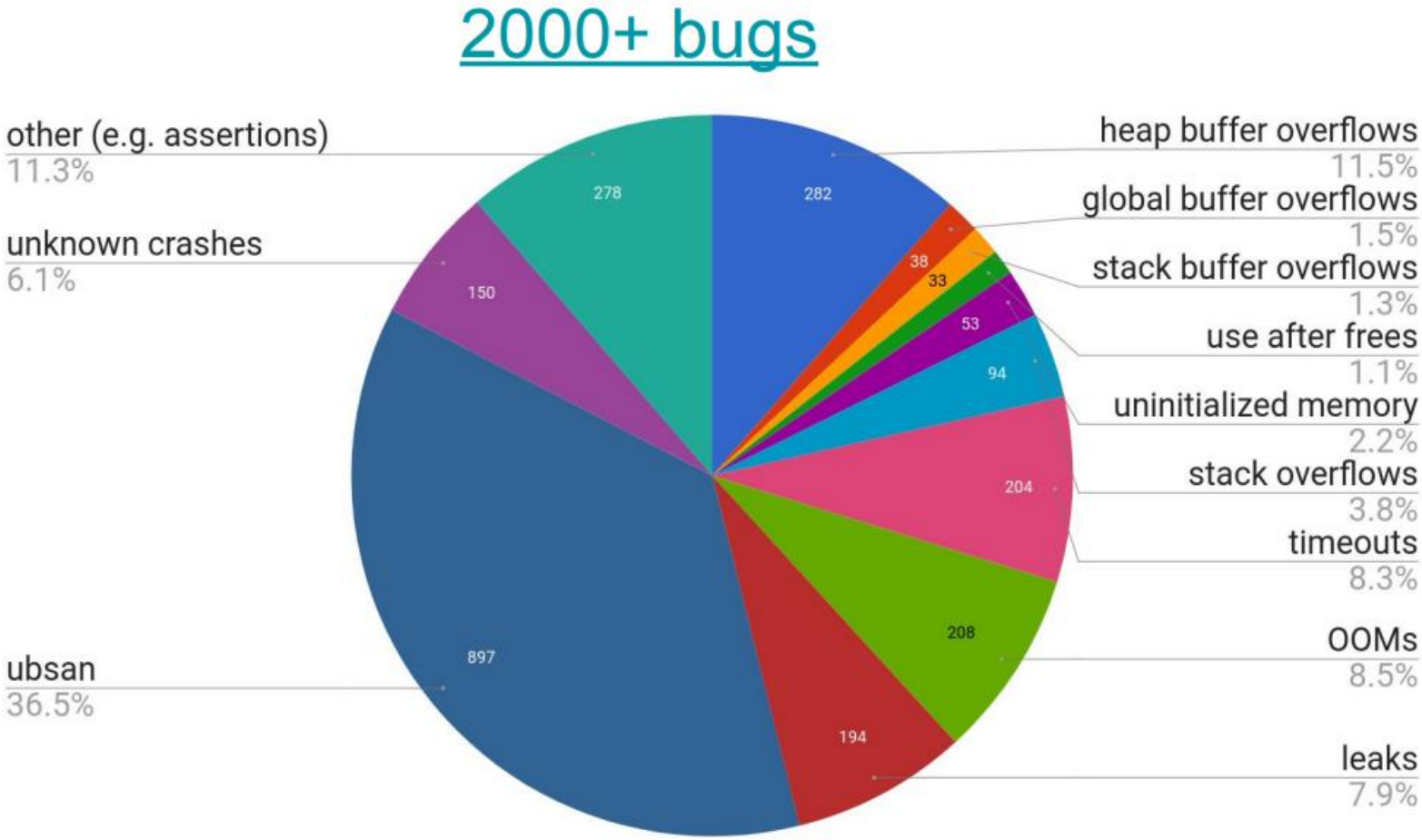
Fuzzing - advantages/disadvantages

- 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 are 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://msrc.microsoft.com/blog/2017/10/vulnscan-automated-triage-and-root-cause-analysis-of-memory-corruption-issues/

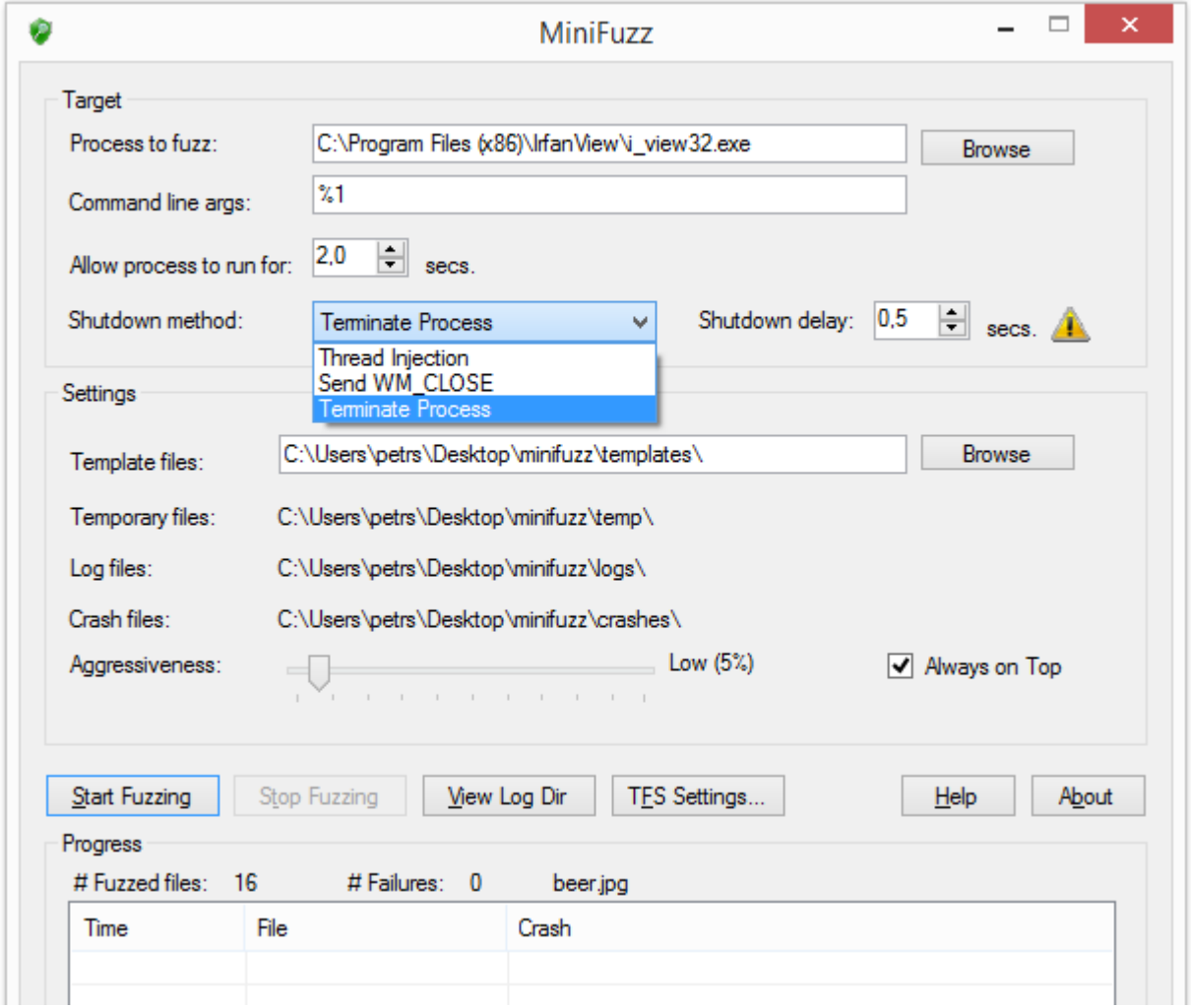
What kind of bugs are usually missed?

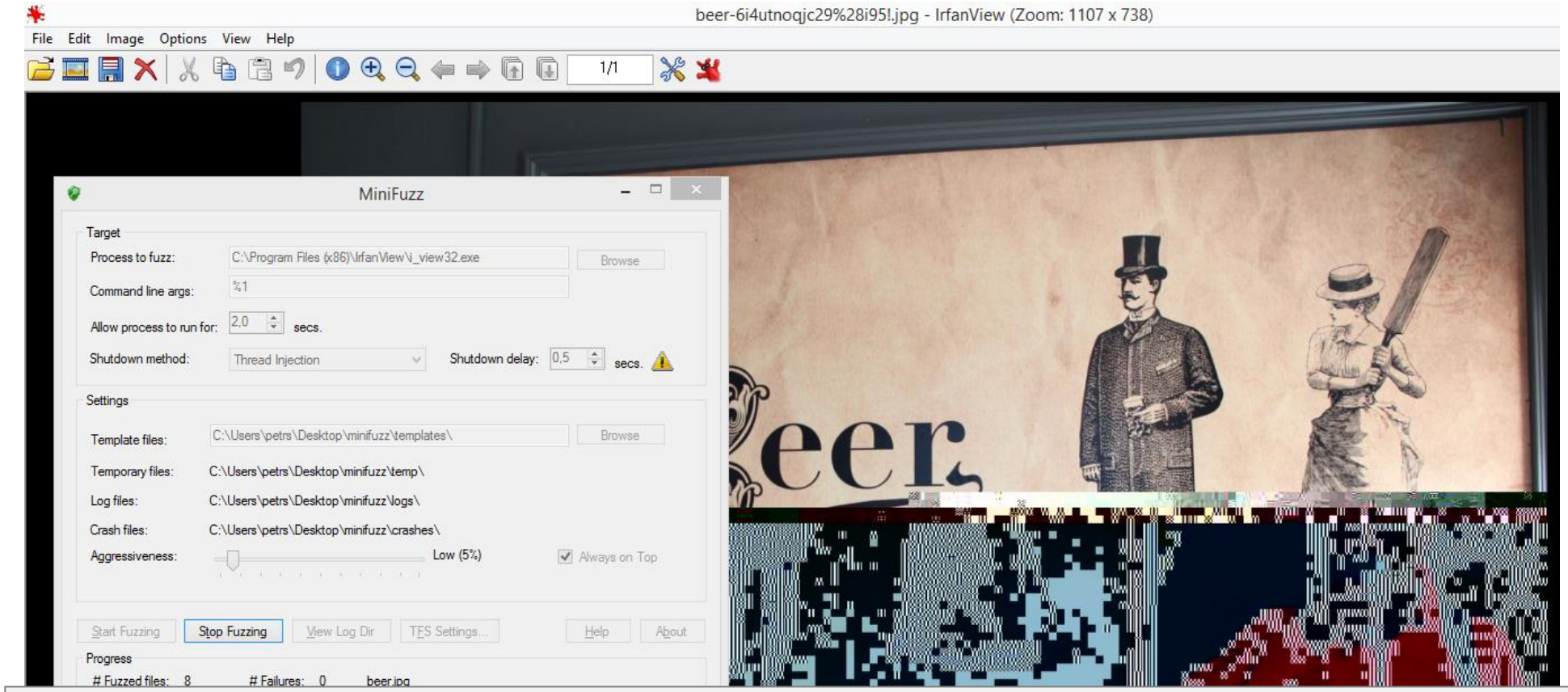
- Bugs after input validation (if not modeled properly)
- High-level / architecture bugs (e.g. weak crypto)
- Usability bugs
- …

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

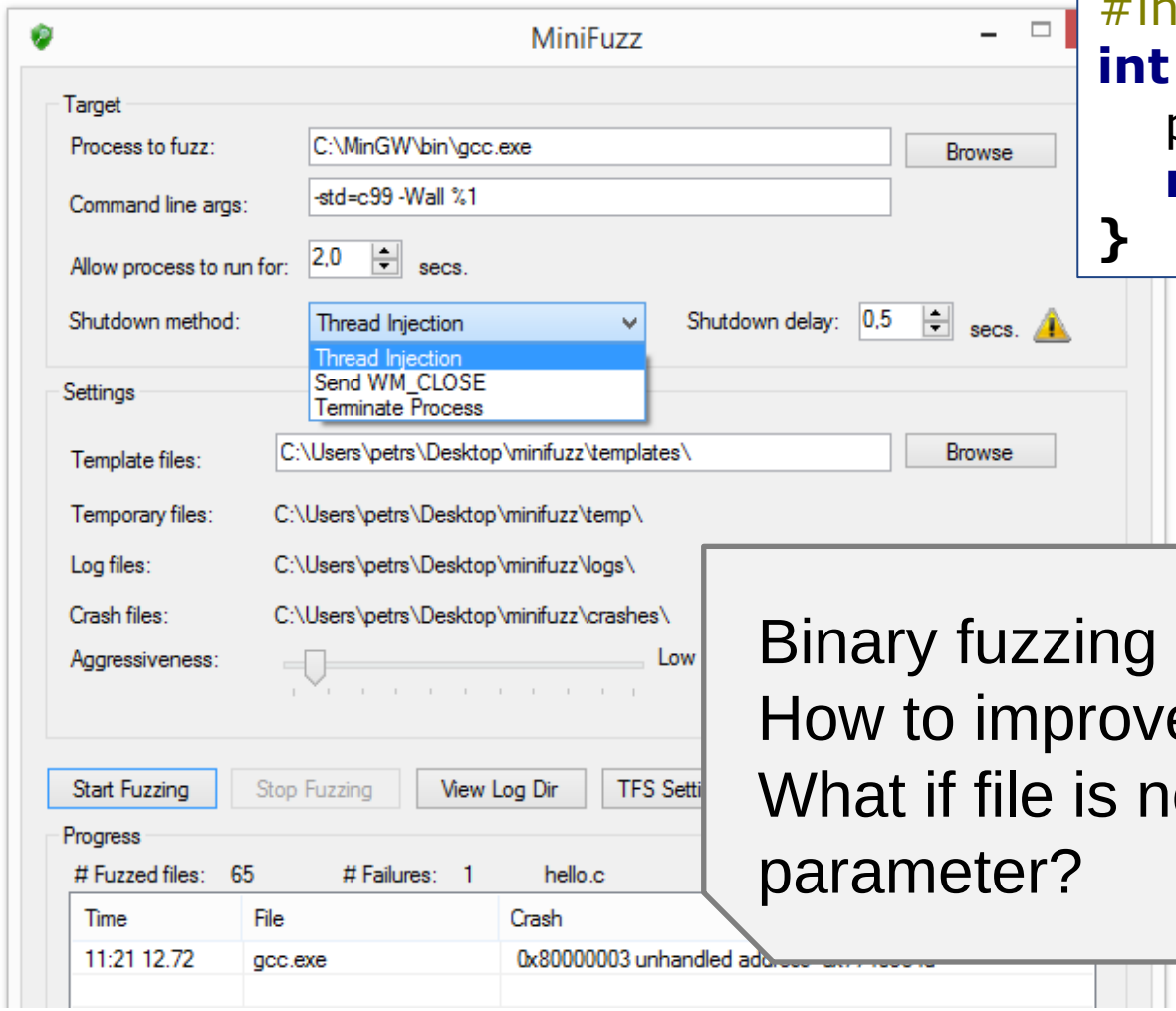




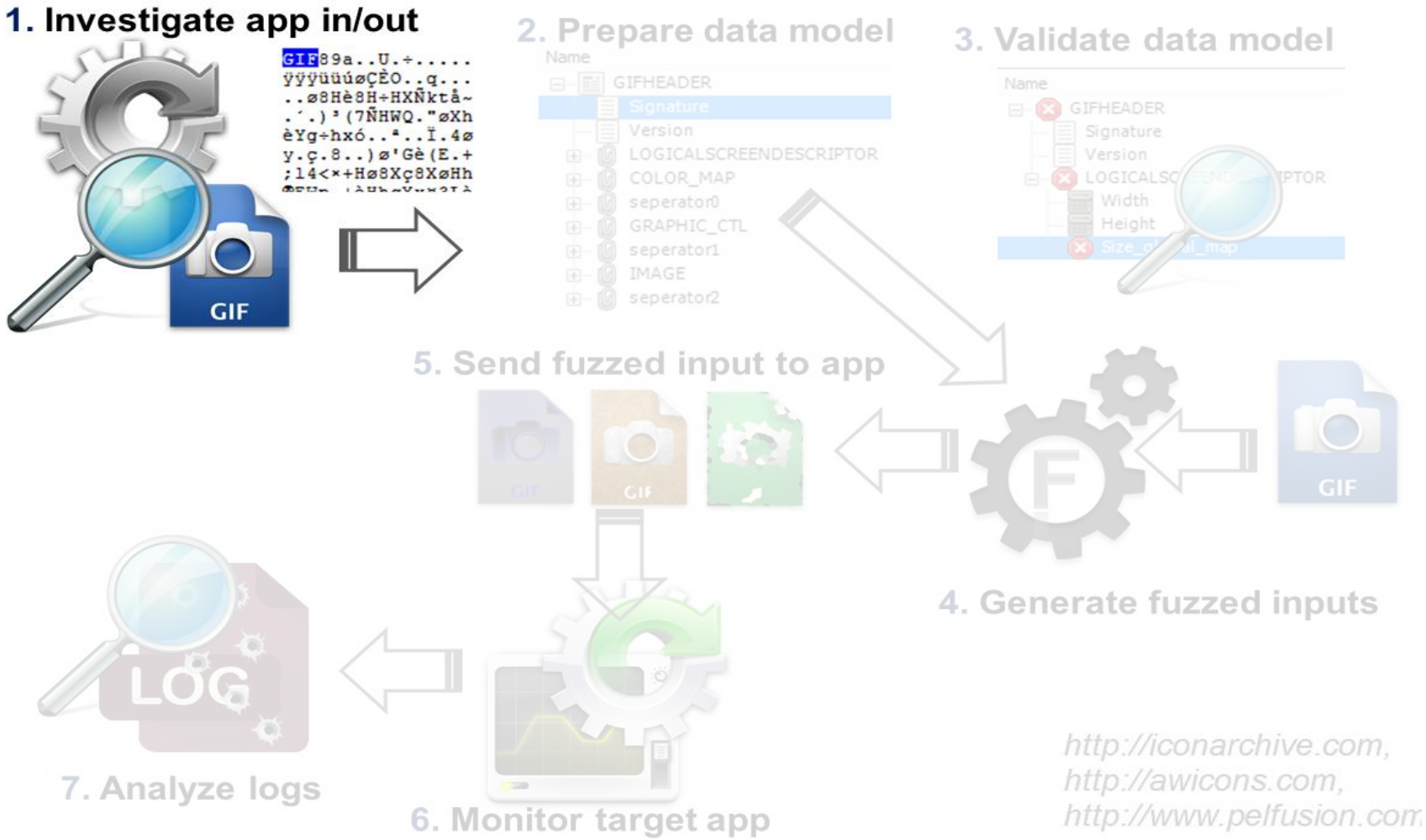

```
<?xml version="1.0"?>
<failures>
  <failure type="Exception Event:Tid=8504, 0x80000003, unhandled, address=0x7740e34d" datetime="11:21:12 12. 2. 2015">
    <registers RAX="00000000" RBX="00000000" RCX="7FFF5FC5180A" RDX="00000000" RSI="00000000" RDI="00000000" RBP="00000000" RSP="00
    <process 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;
}
```



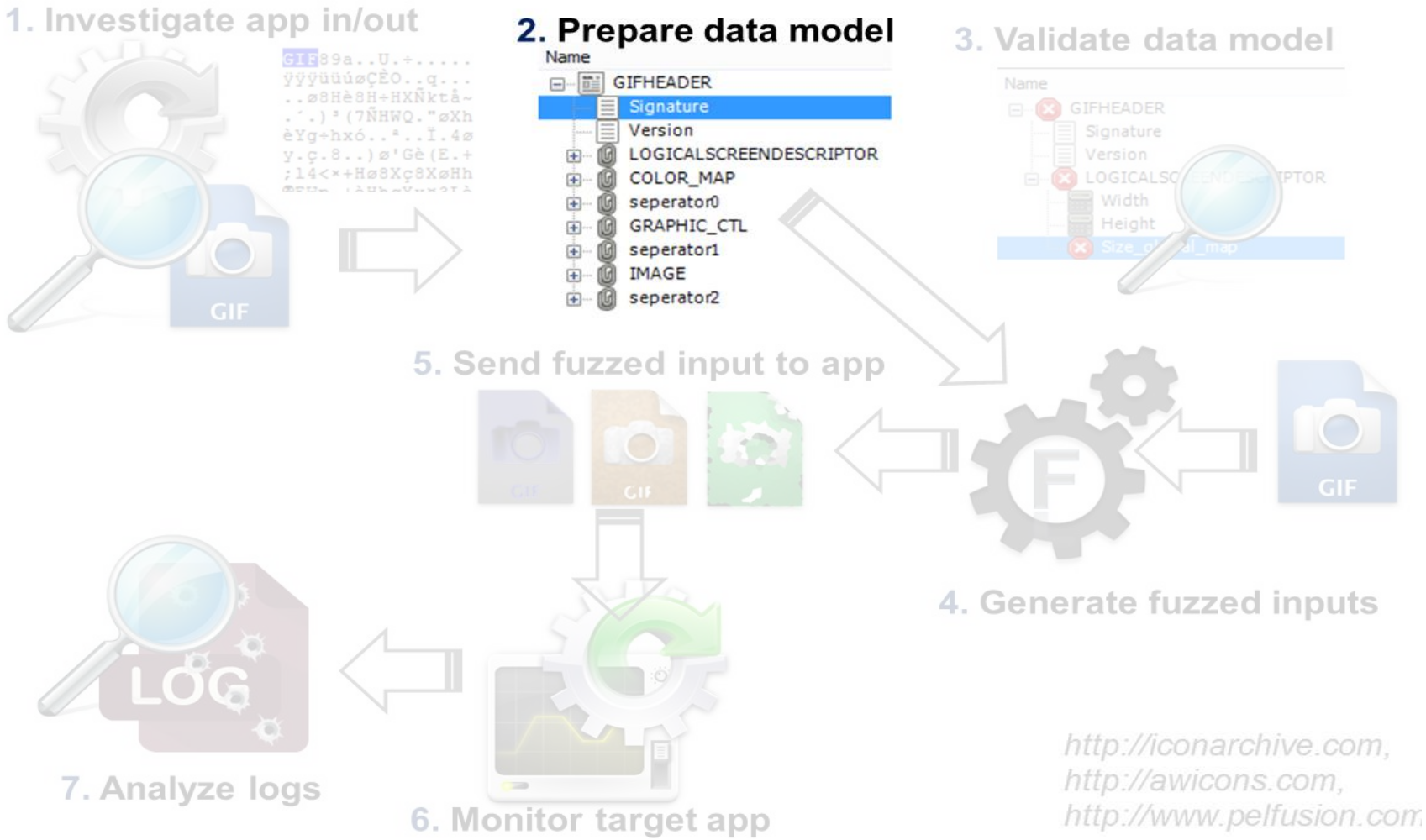
Binary fuzzing of source code???
How to improve test coverage?
What if file is not command line parameter?



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
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,
http://www.pelfusion.com

MODELLING

Input preparation

- *Time intensive part of fuzzing (if model does not exist 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

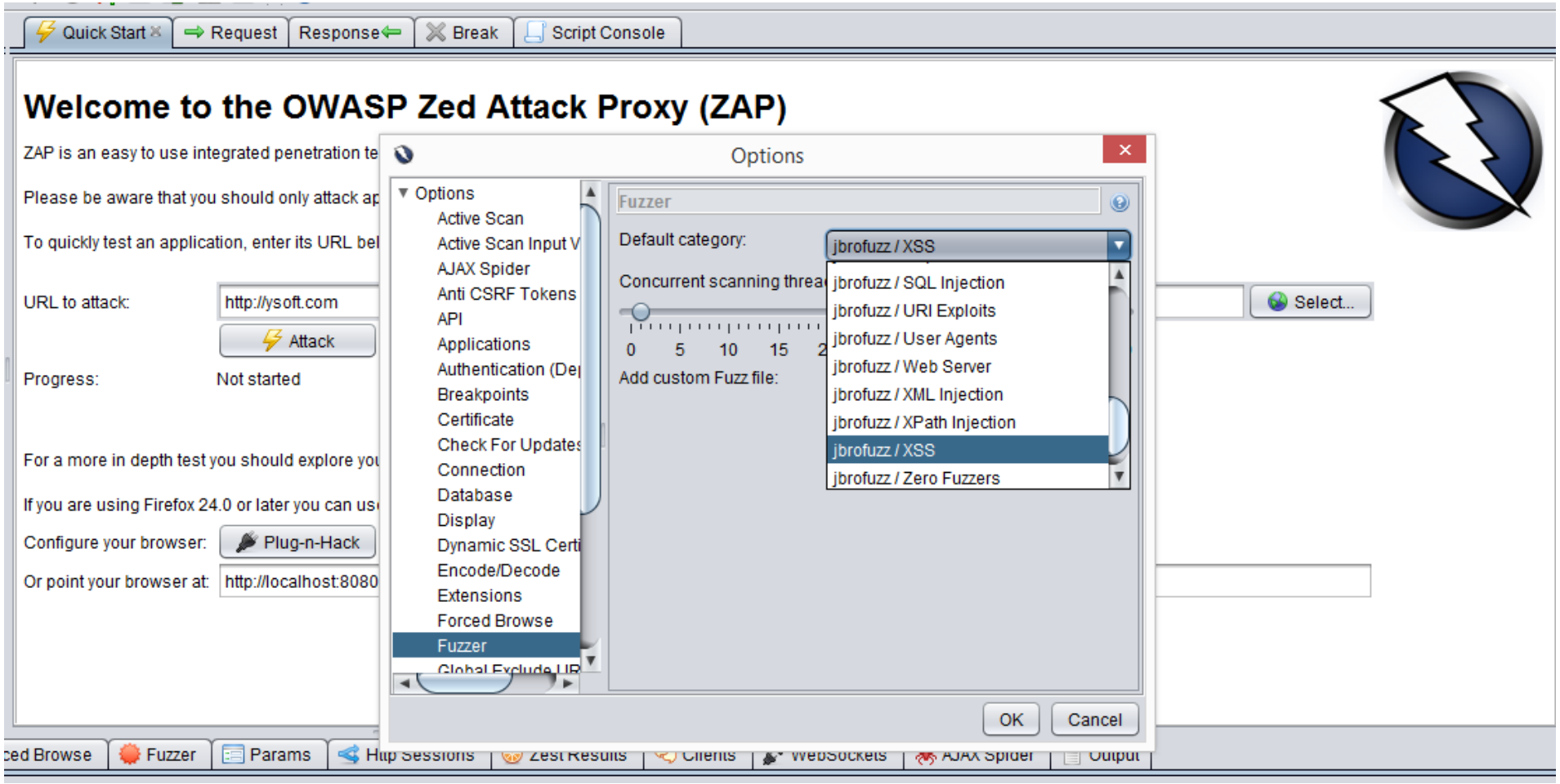
- *"…easy-to-set-up general purpose shotgun test to expose the easiest cracks…"*
  - https://gitlab.com/akihe/radamsa
- Just provide input files, all other settings automatic
  - **cat** file **|** radamsa **>** **file.fuzzed**

```
>echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
1 + (2 + (2 + (3 + 4?)
1 + (2 + (3 +?4))
184467440737095516l5 + 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



Differential fuzzing

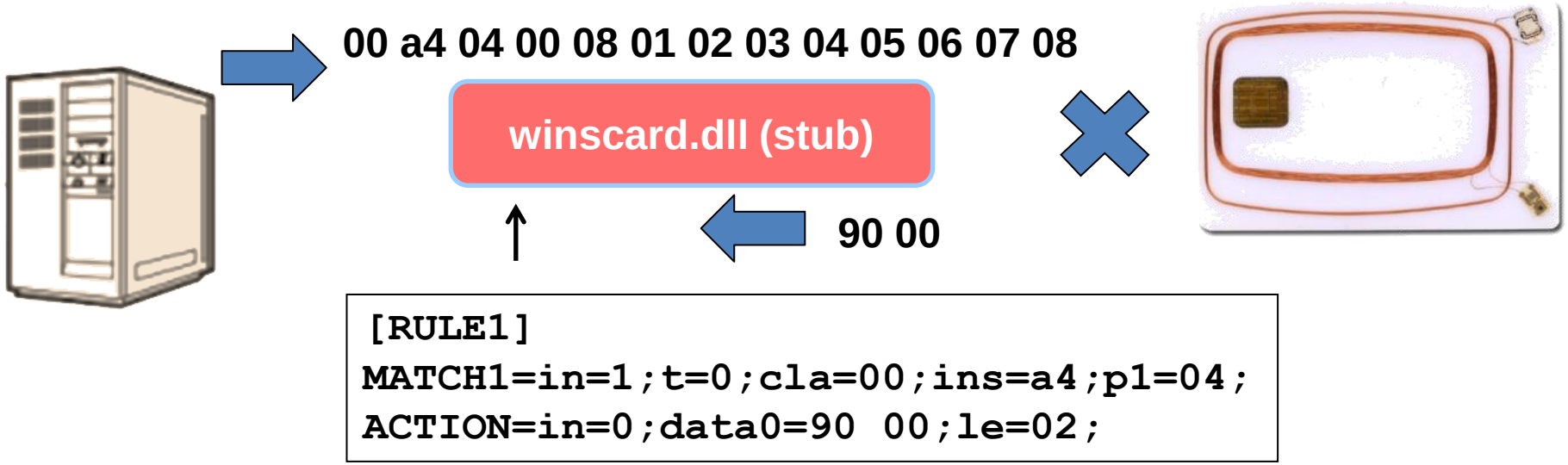
- Basic idea
  - Compare results obtained from two (or more) implementations for the same inputs
- Usage scenarios
  - Legacy and refactored implementation (additional check atop of unit tests)
  - Conformance of independent implementation to the reference one
  - Comparison of expected outputs from group of programs
- Solves the issue of missing expected outputs (insufficient test vectors)
  - Expected behavior is taken from the other program (reference, majority)

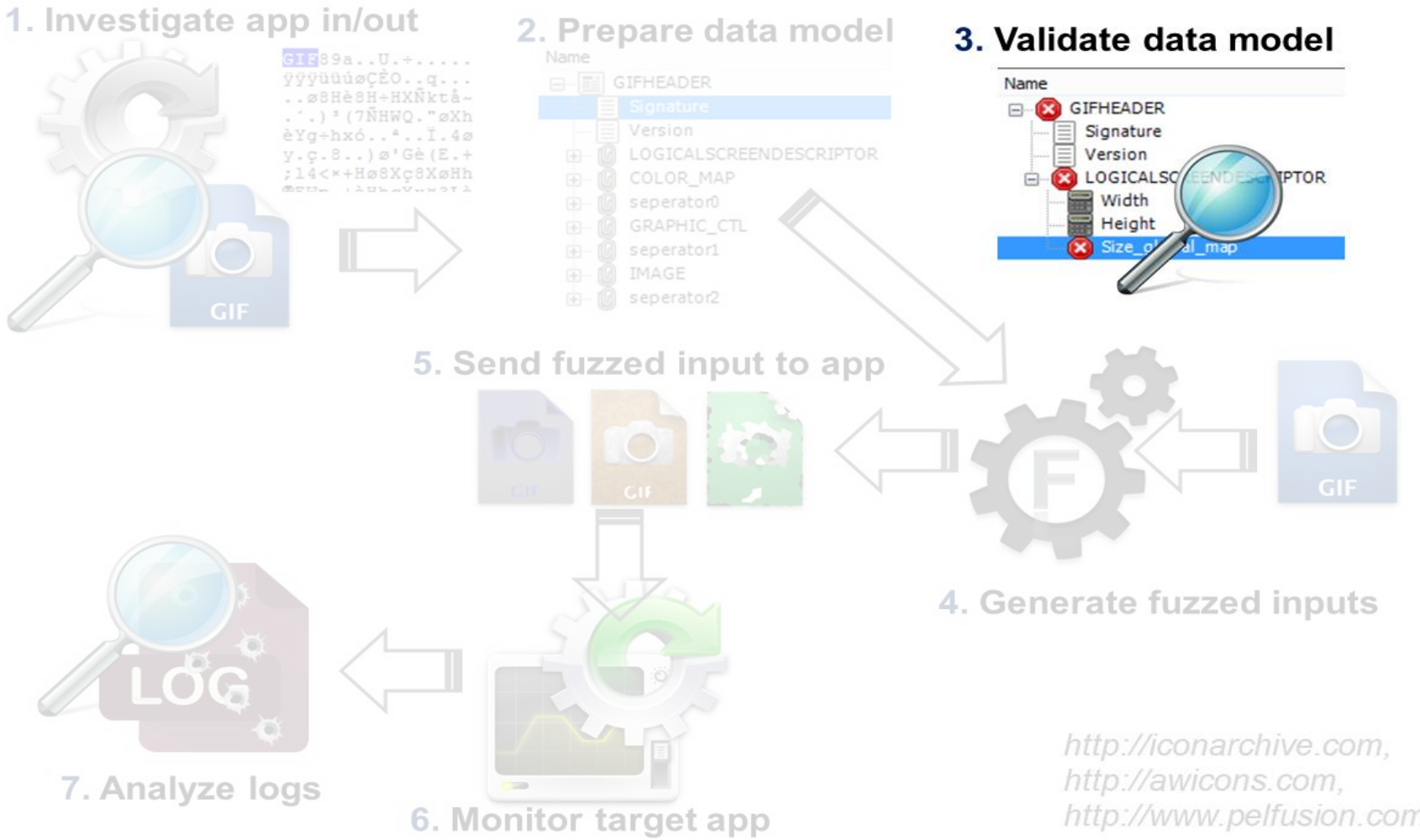
Fuzzing in cryptographic domain

- CryptoFuzz: differential fuzzing of cryptographic libraries
  - https://github.com/guidovranken/cryptofuzz
  - Provides same input to multiple cryptographic libraries, compare outputs
  - The “correct” result is the one by majority of libraries
- TLS fuzzer
  - https://github.com/tomato42/tlsfuzzer/
  - Verifies correct error handling by TLS server (expected error message)
  - Incorrect error behavior can lead to decryption of data or private key extraction (padding oracle attacks, e.g., https://robotattack.org/)

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
  - …



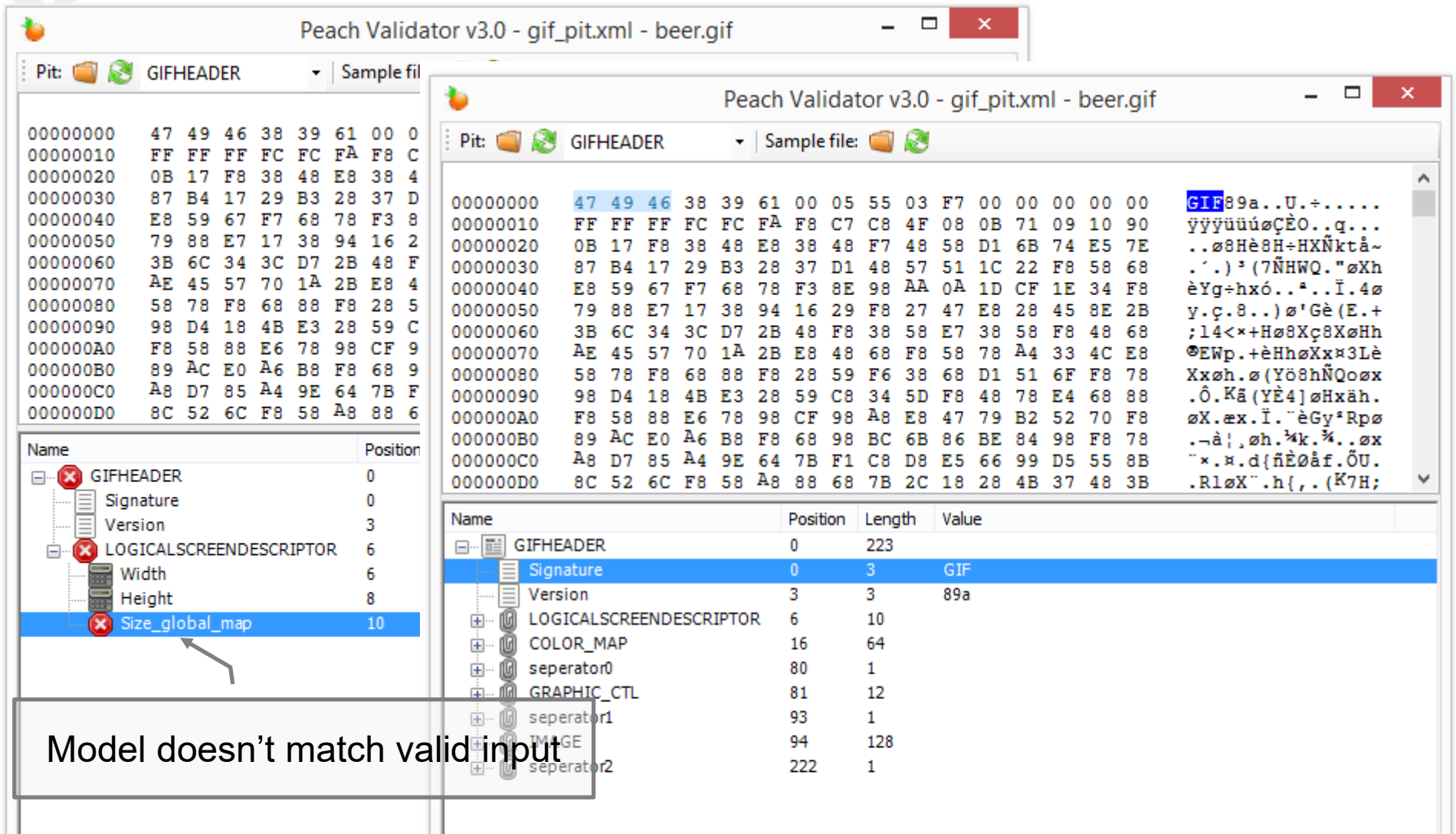

1. Investigate app in/out
GIF
2. Prepare data model
Name
GIFHEADER
Signature
Version
LOGICALSCREENDESCRIPTOR
COLOR_MAP
seperator0
GRAPHIC_CTL
seperator1
IMAGE
seperator2
3. Validate data model
Name
GIFHEADER
Signature
Version
Width
Height
4. Generate fuzzed inputs
GIF
5. Send fuzzed input to app
6. Monitor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

VALIDATION

Validation of model

- Are fuzzed inputs according to your need?
  - Smarter fuzzing understands a data format
  - Wrong data format usually fails early on initial parsing
- Check between fuzzing data model and real input
  - E.g., Peach Validator tool
- Are template files providing good test coverage?
  - E.g., Peach minset tool

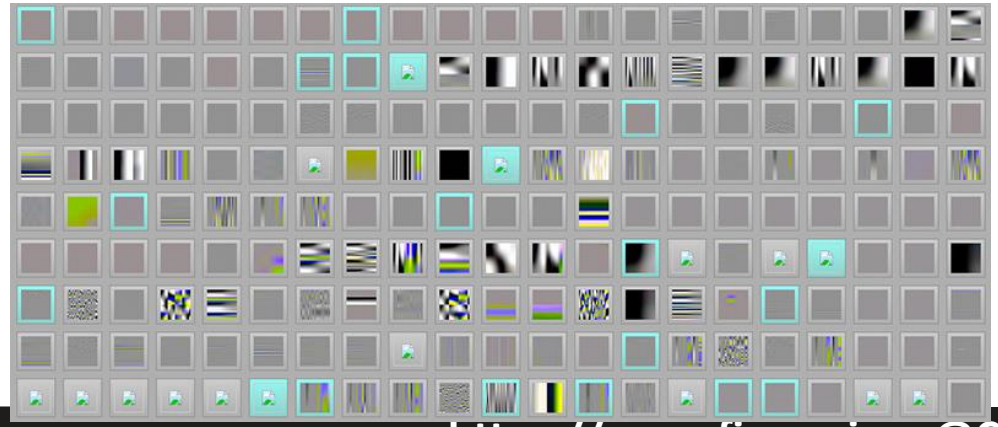
Peach Validator 3.0



Model doesn't match valid input

American fuzzy lop

- State of the art and very powerful tool by Google
- 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

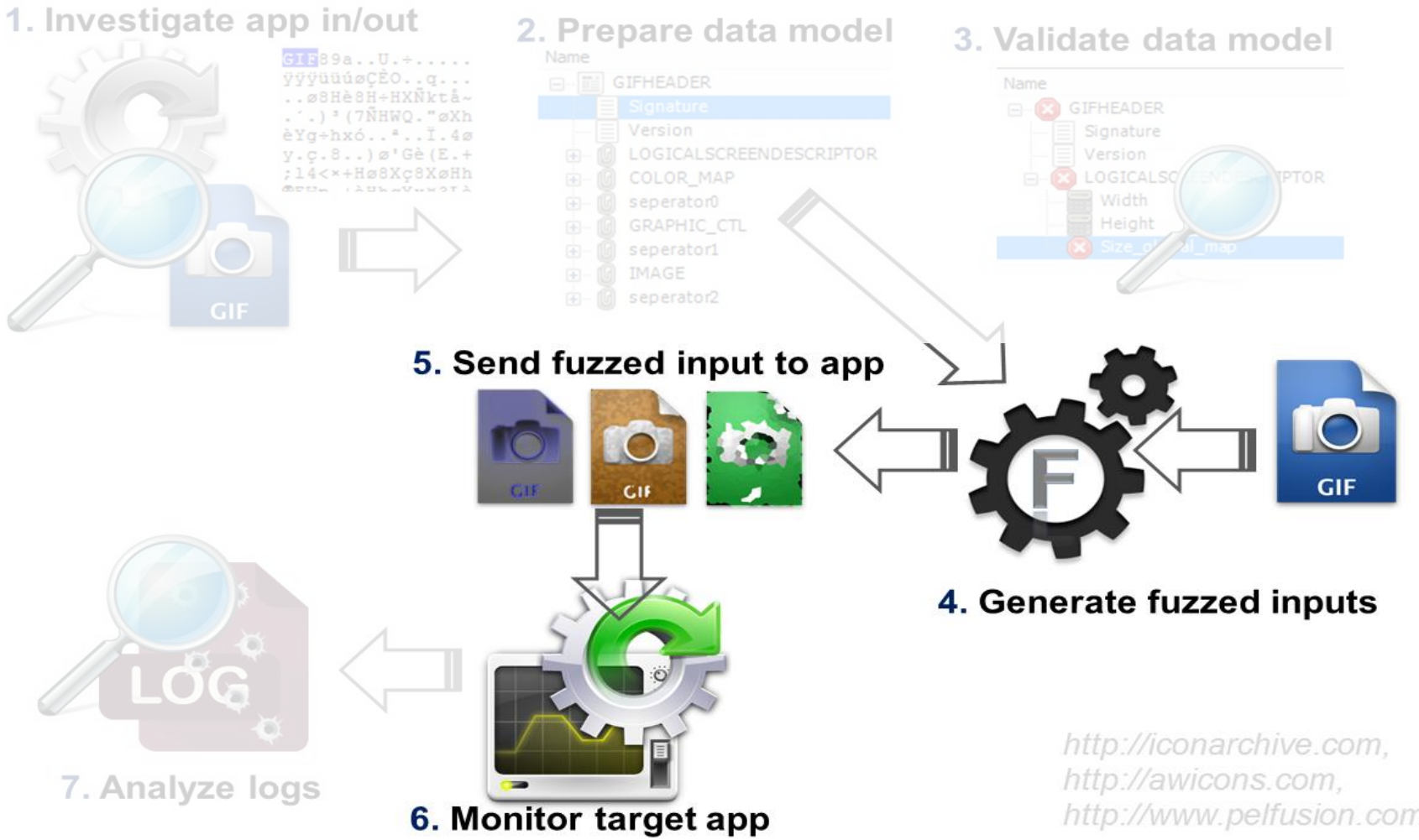


American Fuzzy Lop plus plus

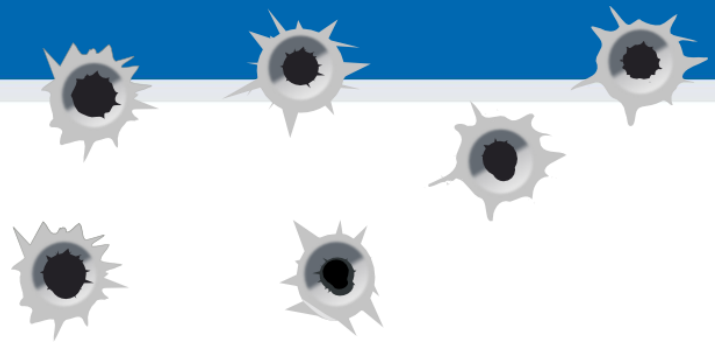
- Relative inactivity of Google's upstream AFL development since 2017.
- Result: AFL++, a fork to Google's AFL aiming at:
  - more speed,
  - more and better mutations,
  - more and better instrumentation,
  - custom module support, etc.
  - AFL & AFL++: https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer)
- Links:
  - https://aflplus.plus/
  - https://github.com/AFLplusplus/AFLplusplus
- Google's OSS-Fuzz initiative, which provides free fuzzing services to open-source software, replaced AFL with AFL++ in 2021.

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 & AFL++ fuzzers use compile-time instrumentation + genetic programming to create test cases



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
- …

GitLab Acquires Peach Tech and Fuzzit to Expand its DevSecOps Offering

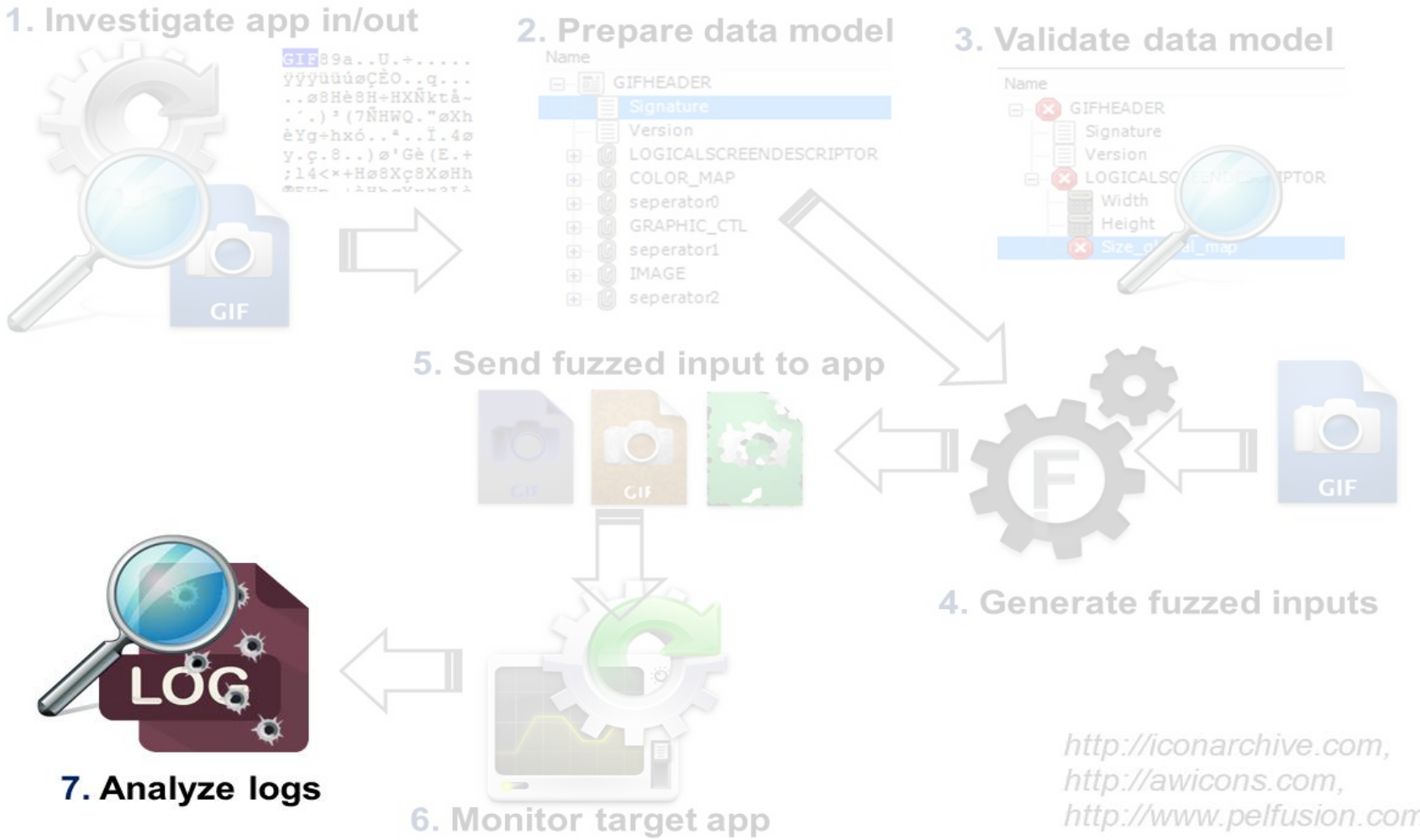
Acquisitions will make GitLab the first security solution to offer both coverage-guided and behavioral fuzz testing

Acquisitions will make GitLab the first security solution to offer both coverage-guided and behavioral fuzz testing

On this page

SAN FRANCISCO, CALIFORNIA — June 11, 2020 -

More at: https://peachtech.gitlab.io/peach-fuzzer-community/


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,
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 are less relevant
  - Crash analyzer: https://learn.microsoft.com/en-us/microsoft-desktop-optimization-pack/dart-v7/diagnosing-system-failures-with-crash-analyzer--dart-7
  - !exploitable https://msecdbg.codeplex.com/ (not available anymore)
- Hits reproduction
  - Hit can be the result of cumulative series of operations
- Many hits are duplicates
  - Inputs are different but hit caused in the same part of the code
- (Automatic) Bucketing of hits
  - E.g., Peach performs bucking based on the signature of callstack

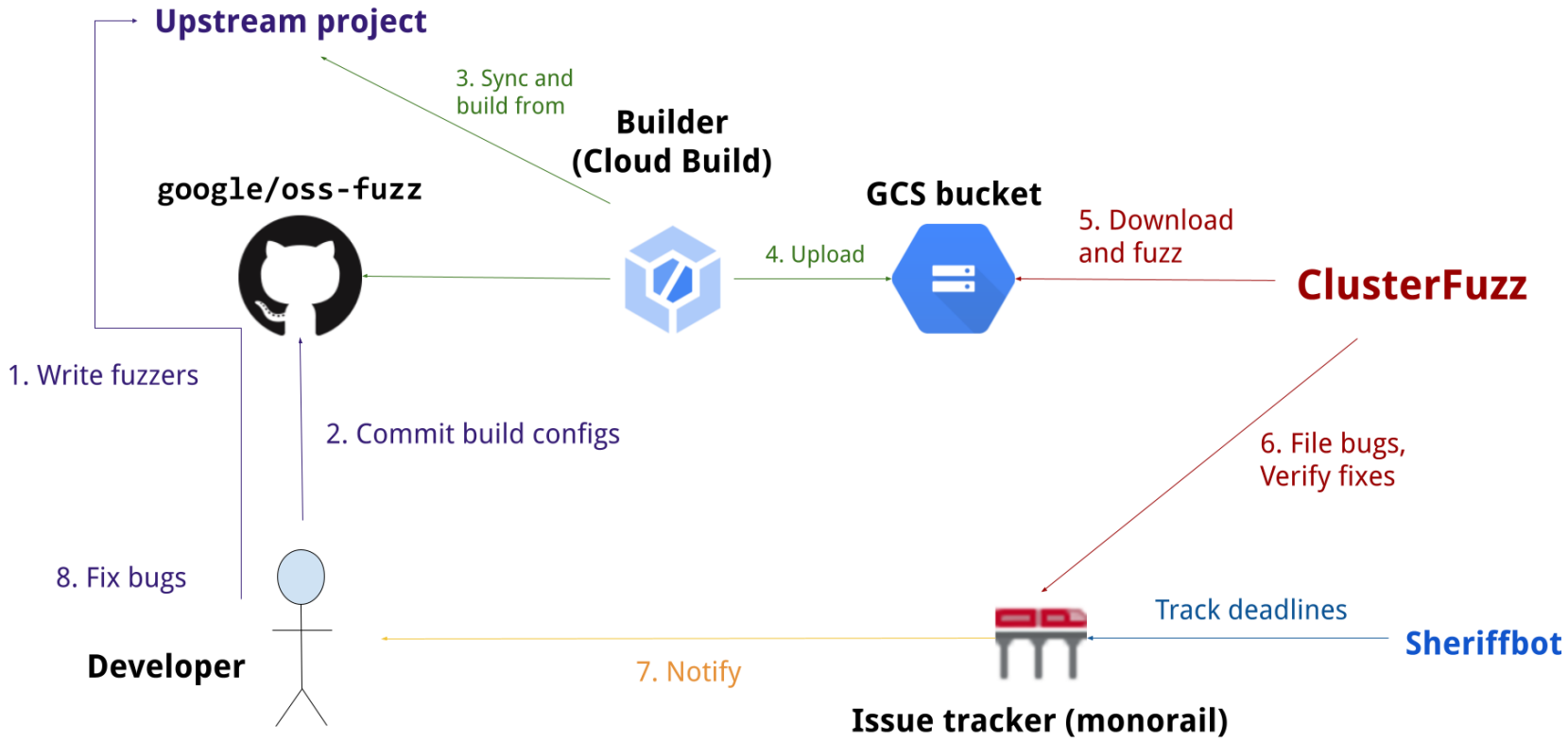
Summary for fuzzing

- 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!
- The Top Fuzzing Open-Source Projects (23 in 2024)
  - https://awesomeopensource.com/projects/fuzzing

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 OSS projects on Google’s servers
  - Can be replicated in your Continuous Integration server

Google OSS-Fuzz:
Continuous Fuzzing for Open Source Software



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*
- **Sinks** – attacked final functionality, e.g. system calls
- Native support in some languages (Ruby, Perl)
  - But not C++/Java ☹, FindSecurityBugs adds taint analysis 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 a “cheap” blackbox approach via malformed inputs
- Mandatory reading/watching
  - 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
  - https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

Questions ?