Finding security vulnerabilities by fuzzing and dynamic code analysis
Security Vulnerabilities

Top code security vulnerabilities don’t change much:
Security Vulnerabilities

Top code security vulnerabilities are still:

Buffer Overflow

Improper Input Validation (inc. Command Injection)

Resource management (inc. Dynamic Memory problems)

Leading to:

modification of program data

change of program control flow

arbitrary code execution
Security Vulnerabilities

Buffer Overflow: Often due to lack of bounds checking in C

```c
void foo (void) {
    char buffer[8];
    fill (buffer);
    process (buffer);
    output (buffer);
}

void fill (char *p_buffer) {
    gets (p_buffer);
}
```

1. No protection against writing more than 8 chars

2. This stack frame corrupted
Security Vulnerabilities

Top code security vulnerabilities are still:
Buffer Overflow, Command Injection, Memory Access

Because:
We don't sanitise our inputs
No bounds checking in C
We don't test all code paths
Bad code often hard to locate
Security Vulnerabilities

Buffer overflow often hard to locate:

void foo (void) {
    char buffer[8];
    fill (buffer);
    process (buffer);
    output (buffer);  
}

void fill (char *p_buffer) {
    gets (p_buffer);  
}

1. Buffer overflows here
2. No effect until parent exits if at all
Security Vulnerabilities

Default run-time protection may not help much

```
gcc -fstack-protector -o foo foo.c
```

Enter Buffer: overflowed
Filling buffer 'overflowed'
Processing Buffer 'overflowed'
Output Buffer 'overflowed'
*** stack smashing detected ***:./buffer-foo terminated
Address Sanitizer

Compilers can now add Dynamic Code Analysis

AddressSanitizer detects C/C++ runtime memory errors:

- Use after free (dangling pointer dereference)
- Heap/Stack/Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Compiler option for LLVM/clang (> v3.1) and GCC (> v4.8)
Address Sanitizer

ASan detects the error as soon as it happens

```
clang -g -fsanitize=address -o foo foo.c
```

Enter Buffer: **overflowed**

```
ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 11 at 0x7ffd6d5cf288 thread T0
#0 0x443d3d in fill /tmp/foo.c:6
```

…followed by detailed information and memory map…
ASan identifies the source line of affected memory and the bad code

```c
void foo (void) {
    char buffer[8];
    fill (buffer);
    process (buffer);
    output (buffer);
}

void fill (char *p_buffer) {
    gets (p_buffer);
}
```

This stack frame corrupted

Buffer overflows here
Address Sanitizer

Trade-off between coverage and performance:

- `f stack-protector`
  About 5% performance penalty
  OK for well tested production code

- `f sanitizer=address`
  About 100% performance penalty, OK for testing
Security Vulnerabilities

Top code security vulnerabilities are still:
Buffer Overflow, Command Injection, Memory Access

Because:
We don't sanitise our inputs
No bounds checking in C
We don't test all code paths – Use Fuzzing?
Bad code often hard to locate – Use Compiler Sanitizers
What is Fuzzing?

A stormy night in 1988

Barton Miller logs in from home using a 1200 baud modem

The noisy line alters the Unix commands he types

Some of the random changes cause programs to crash

“And more surprising to me were the programs that were crashing -- common Unix utilities that we all use everyday”

He started a student project to build a “fuzz generator”
What is Fuzzing?

The Fuzz Generator worked well

...they succeeded well beyond my expectations. As we reported in the first fuzz paper [1990], they could crash or hang between 25-33% of the utility programs on the seven Unix variants that they tested.
The Fuzz Generator showed

Random input creates new test cases that developers haven’t considered

= Better test case coverage
Fuzzing Works

- You can fuzz binary code
- You can fuzz hardware
- You can fuzz at scale:
  
  FFmpeg and a thousand fixes
2014: FFmpeg and a thousand fixes

Google blog: 1,120 bugs in ffmpeg mostly found by fuzzing
10% - 20% of them exploitable security vulnerabilities
Fuzzing Works

2014: FFmpeg and a thousand fixes

Google blog: 1,120 bugs in ffmpeg mostly found by fuzzing

10% - 20% of them exploitable security vulnerabilities

• Test suite + 7,000 media files from the Internet
• 200 cores making random changes
• Several dozen iterations over 2 years

No small project!
Fuzz data

Fuzzing normally starts with good test data and makes random changes:

- Bit flip
- Byte swap
- Extend or truncate input
- Maximum, minimum values
Purely random changes can be inefficient

Data may not get past validation checks, like CRC

- Save and reuse random data that does penetrate the code
- Fixup data (e.g. CRC) after randomisation
- Compile without validation
- Compile with permissive stub libraries
- Fuzz layers separately, e.g. Fuzzed HTTP in valid IP packets
- Specialist fuzzers for common protocols
Intelligent Fuzzing

2015: American Fuzzy Lop

- Tries various different input mutation strategies
- Instruments the code to detect code paths exercised
- Remembers data that hits new code paths

= Better code coverage
### American Fuzzy Lop 2.13b (foo)

#### Process Timing
- **Run time**: 0 days, 0 hrs, 0 min, 10 sec
- **Last new path**: None yet (*odd, check syntax!*)
- **Last uniq crash**: 0 days, 0 hrs, 0 min, 9 sec
- **Last uniq hang**: None seen yet

#### Cycle Progress
- **Now processing**: 0 (0.00%)
- **Paths timed out**: 0 (0.00%)

#### Stage Progress
- **Now trying**: havoc
- **Stage execs**: 1066/5000 (21.32%)
- **Total execs**: 41.6k
- **Exec speed**: 4120/sec

#### Fuzzing Strategy Yields
- **Bit flips**: 0/32, 0/31, 0/29
- **Byte flips**: 0/4, 0/3, 0/1
- **Arithmetics**: 0/222, 0/0, 0/0
- **Known ints**: 0/24, 0/83, 0/44
- **Dictionary**: 0/0, 0/0, 0/0
- **Havoc**: 1/40.0k, 0/0
- **Trim**: 20.00%/1, 0.00%

#### Overall Results
- **Cycles done**: 8
- **Total paths**: 1
- **Uniq crashes**: 1
- **Uniq hangs**: 0

#### Map Coverage
- **Map density**: 3 (0.00%)
- **Count coverage**: 1.00 bits/tuple

#### Findings in Depth
- **Favored paths**: 1 (100.00%)
- **New edges on**: 1 (100.00%)
- **Total crashes**: 2087 (1 unique)
- **Total hangs**: 0 (0 unique)

#### Path Geometry
- **Levels**: 1
- **Pending**: 0
- **Pend fav**: 0
- **Own finds**: 0
- **Imported**: n/a
- **Variable**: 0

---

CPU: 193%
Security Vulnerabilities

Top two code security vulnerabilities are still:

Command Injection
Buffer Overflow

Because:
We don't sanitise our inputs
No bounds checking in C
We don't test all code paths – **Use Intelligent Fuzzing**
Bad code often hard to locate – **Use Compiler Sanitizers**
A Winning Combination

A typical example: 13 security vulnerabilities reported...

<table>
<thead>
<tr>
<th>Subject</th>
<th>From</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[oss-security] audiofile: multiple crashes</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:43</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in MSADPC...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:45</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in readValu...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:46</td>
</tr>
<tr>
<td>[oss-security] audiofile: global buffer overflow in decodeSample...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:49</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in alaw2lin...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:50</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in IMA::dec...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:52</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in MSADPC...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:53</td>
</tr>
<tr>
<td>[oss-security] audiofile: divide-by-zero in BlockCodec::runPull (B...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:53</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in ulaw2lin...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:54</td>
</tr>
<tr>
<td>[oss-security] audiofile: divide-by-zero in BlockCodec::reset1 (B...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:55</td>
</tr>
<tr>
<td>[oss-security] audiofile: heap-based buffer overflow in Expand3...</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:55</td>
</tr>
<tr>
<td>[oss-security] audiofile: multiple ubsan crashes</td>
<td>Agostino Sarubbo</td>
<td>26/02/17 11:56</td>
</tr>
</tbody>
</table>
A Winning Combination

...found with AFL and Sanitisers

Description:

audiofile is a C-based library for reading and writing audio files in many common formats.

A fuzz with a wav file as input produced an heap overflow.

The complete ASan output:

==6096==ERROR: AddressSanitizer: heap-buffer-overflow...

...

Note:

This bug was found with American Fuzzy Lop.
A Winning Combination

Intelligent Fuzzers create test data for

• Create test data you hadn’t thought of
• Improve code path test coverage
• E.g. AFL, libfuzz

Compiler Sanitisers isolate the problem

• AddressSanitizer (detects addressability issues)
• LeakSanitizer (detects memory leaks)
• ThreadSanitizer (detects data races and deadlocks) for C++ and Go
• MemorySanitizer (detects use of uninitialized memory)
• UndefinedBehaviourSanitizer (detects conversion and overflow problems)
Fuzzing Services

Google OSS-Fuzz
Continuous fuzzing of important open source code

Microsoft Project Springfield
Fuzz testing service
AddressSanitizer, Clang 5 Documentation, http://clang.llvm.org/docs/AddressSanitizer.html

Sanitizers project home, Google, https://github.com/google/sanitizers

American Fuzzy Lop: http://lcamtuf.coredump.cx/afl/

Basic Fuzzing Framework, CERT, http://www.cert.org/download/bff/, (BFF is based on zzuf)


FFmpeg and a thousand fixes, Google blog, http://j00ru.vexillium.org/?p=2211

The Fuzzing Project, tutorials, fuzz vectors and other resources: https://fuzzing-project.org/

libFuzzer – a library for coverage-guided fuzz testing, LLVM, http://llvm.org/docs/LibFuzzer.html

Google OSS-Fuzz project, https://github.com/google/oss-fuzz

Microsoft Springfield fuzz testing service: https://www.microsoft.com/en-us/springfield/


OWASP fuzzing guide: https://www.owasp.org/index.php/Fuzzing

OWASP ZAP Fuzzer: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

zzuf – Multipurpose Fuzzer: http://caca.zoy.org/wiki/zzuf


Image credits

http://howtorabbit.blogspot.co.uk/2015/02/rabbit-species-american-fuzzy-lop.html

First Place Ribbon: Copyright: snake3d / 123RF Stock Photo, http://www.123rf.com/profile_snake3d
Finding security vulnerabilities by fuzzing and dynamic code analysis

MathEmbedded

- Embedded Security Reviews
- Secure Software Design
- Security Training

www.mathembedded.com