Protocol Fuzzer on Embedded Firmware

A Case Study

Dor Levy, Michael Stahl
QA&Test, Oct'21

Whoami == Dor Levy

- Senior Security Researcher @ Intel
- MSc in computer engineering & applied physics (Hebrew University)
- Issued 20 patents in various fields including security systems and user & autonomous systems & co-authored 10 papers

My co-author: Michael Stahl

- SW Validation Architect @ Intel
- BSc in Electronics Engineering (Ben Gurion U)
- 22 years' experience in testing embedded software
- Papers and presentations: www.testprincipia.com

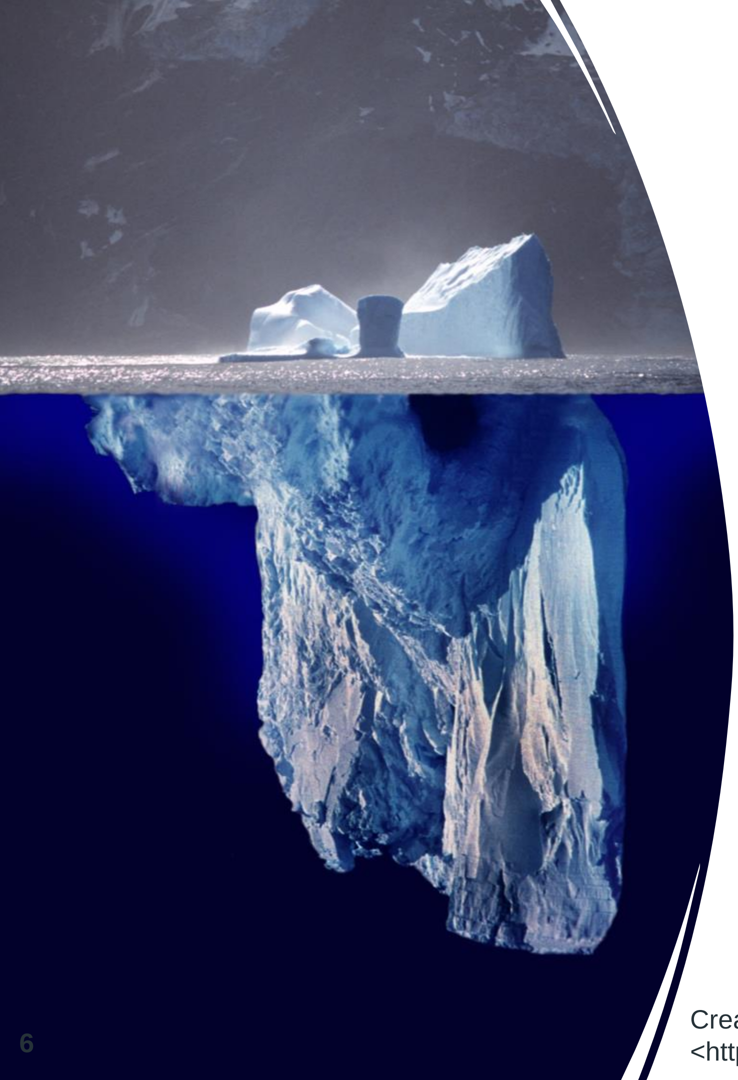
Agenda

- Fuzzing: Concept, terms and definitions
- DUT overview
  - The embedded system
  - The protocol
- Embedded System Fuzzing Challenges
- Fuzzer Architecture
- Results & lessons learned

1.

Fuzzing

Concept, terms and definitions



The challenge of Input Validation

```c
void set_clock_settings(
                 clock_req_t    *pReq,
                 uint32_t*      pRspLen);
```

Input

```
void set_clock_settings(
                clock_req_t   *pReq,
                uint32_t*     pRspLen);
```

```
typedef struct _clock_req_t
{
   clk_header                     Header;
   uint8_t                        ReqClock;
   uint8_t                        SettingType;
    ... (15 more...)
} clock_req_t;
```

Input

Input

```
typedef struct _clock_req_t
{
    clk_header                  Header;
    uint8_t                     ReqClock;
    uint8_t                     SettingType;
     ... (15 more...)
} clock_req_t;
```

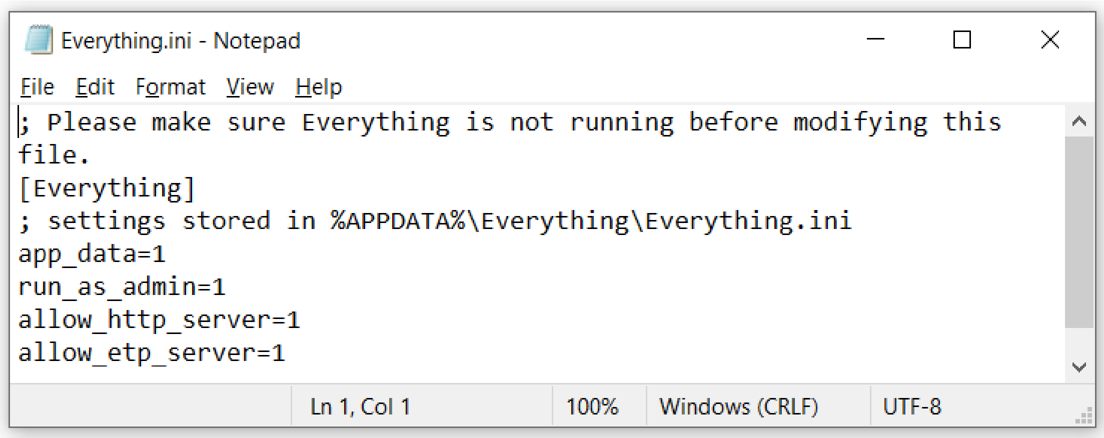
Enumerations (also inputs!)

Another Struct!!! ☹

```
typedef struct _clk_header
{
    uint32_t          ApiVersion;
    COMMAND_ID        CommandId;
    status_t          Status;
    uint32_t          BufferLength;
    CMD_FLAGS         Flags;
} clk_header;
```

Other examples…

- Windows Registry keys
- Each parameter in a config file, INI file, etc



Command line arguments

```
C:\WINDOWS\system32>pict
Pairwise Independent Combinatorial Testing

Usage: pict model [options]

Options:
 /o:N     - Order of combinations (default: 2)
 /d:C     - Separator for values  (default: ,)
 /a:C     - Separator for aliases (default: |)
 /n:C     - Negative value prefix (default: ~)
 /e:file - File with seeding rows
 /r[:N]  - Randomize generation, N - seed
 /c       - Case-sensitive model evaluation
 /s       - Show model statistics
```

◎ Fields in network packet or streams

Table 9 – Control packet header format

	Bits			
Bytes	**31..24**	**23..16**	**15..08**	**07..00**
00..03	MC ID	Header Revision	Reserved	IID
04..07	Control Packet Type	Ch. ID	Reserved	Payload Length
08..11	Reserved			
12..15	Reserved			

- Configurations
- Messages sent via drivers
- Values parsed from a blob
- Sensor data

…

You got the idea.

Motivation:

Improve coverage

Find vulnerabilities

Attackers use vulnerabilities to produce exploits, from denial-of-service through to full remote code execution.

How?

Automatically generate many inputs

Automatically apply them to the DUT

Monitor results

Goal:

- High (combinatorial) coverage

Easier said…

"Automatically generate" – How?

"Monitor results" – what's expected?

Solution: Fuzzing

- SW testing technique using auto-generated inputs
- Input generated by “mutation engines”
- Expected results are “no crashes; no hangs”
- Best fit for testing SW that takes structured inputs (e.g. parsers of formats or protocols)
- Widely used in information & SW security industries

Why should we use it?

- Fully automated process
- Identify potential security vulnerabilities
- Improves coverage
- Relatively easy to start
- Can save your org time and money

Fuzzing-sensitive bugs

Specific C/C++ bugs that require the sanitizers to catch:

- Use-after-free, buffer overflows
- Uses of uninitialized memory
- Memory leaks

Logical bugs:

- Discrepancies between two implementations of the same protocol
- Round-trip consistency bugs (e.g. compress → decompress → compare to original)
- Assertion failures

Fuzzing-sensitive bugs

Arithmetic bugs:

- Div-by-zero, int/float overflows, invalid bitwise shifts

Plain, simple crashes:

- NULL dereferences, Uncaught exceptions

Concurrency bugs:

- Data races, Deadlocks

Resource usage bugs (stress):

- Memory exhaustion, hangs or infinite loops, infinite recursion (stack overflows)

Potential fuzzing targets

- Parsers of any kind (xml, pdf, truetype,...)
- Media codecs (audio, video, vector images, ...)
- Network protocols
- Compression (zip, gzip, ...)
- Compilers; Interpreters (PHP, Perl, Python, ...)
- Regular expression matchers (PCRE, RE2, libc)
- Databases (SQlite)
- Browsers (all)
- Text editors/processors (vim, OpenOffice)
- OS Kernels (Linux), drivers, supervisors, VMS
- UI (Chrome UI)

Etc. etc.

Types of Fuzzers

Input-seed driven

Input-structure driven

Program-structure driven

Input-seed driven

Billing details

Name *

!2fdsafdsa@

Surname *

sdfas$ @#

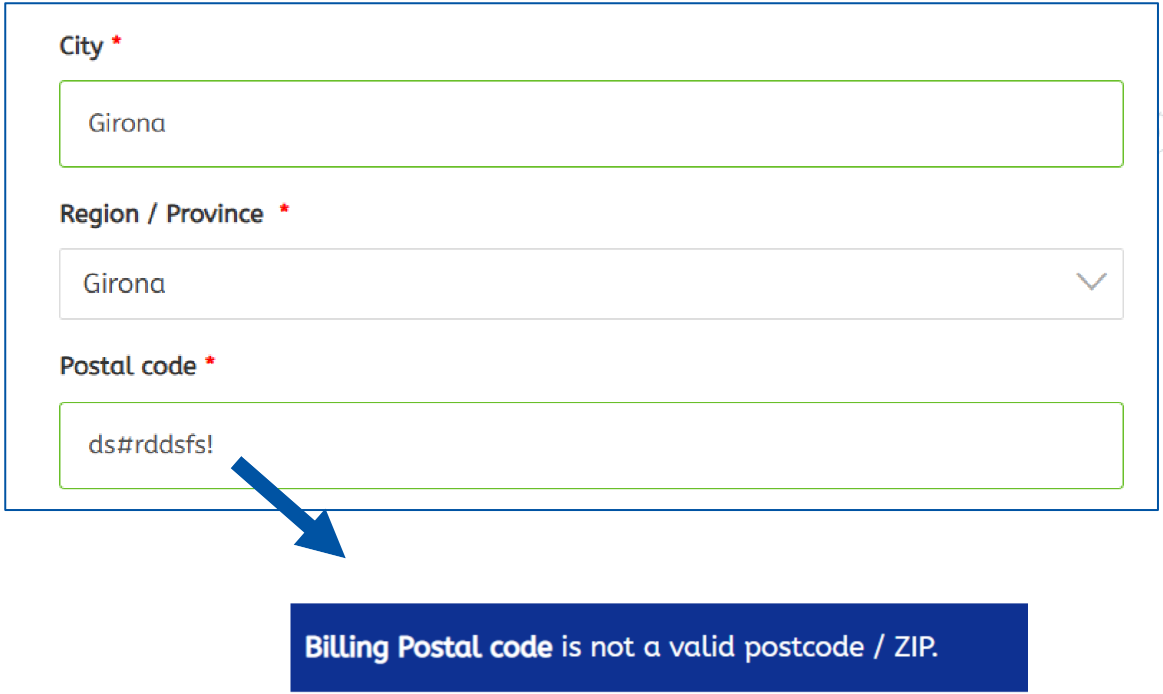
VAT number *

sads313trr43evc

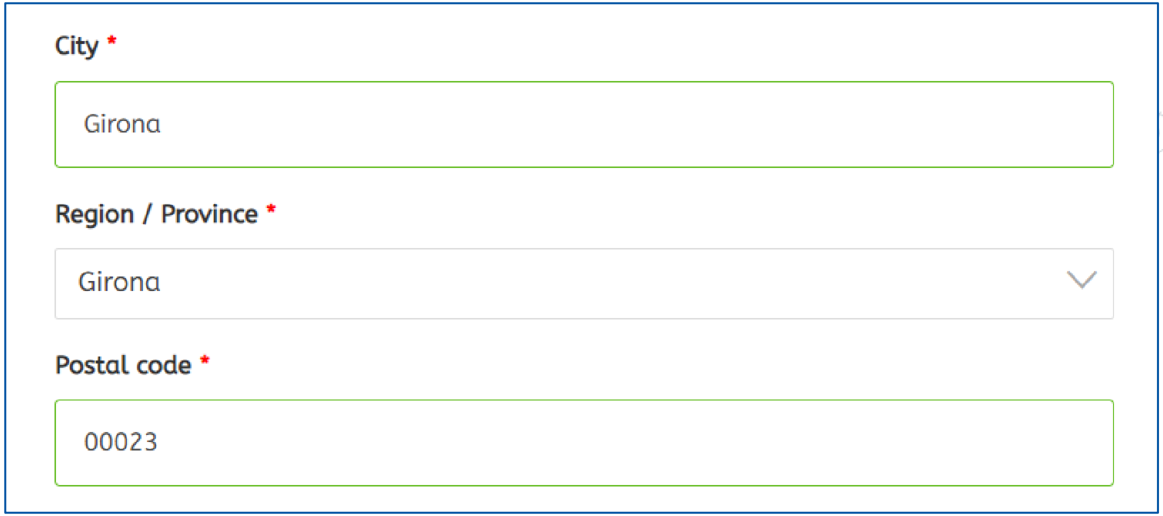
Company (optional)

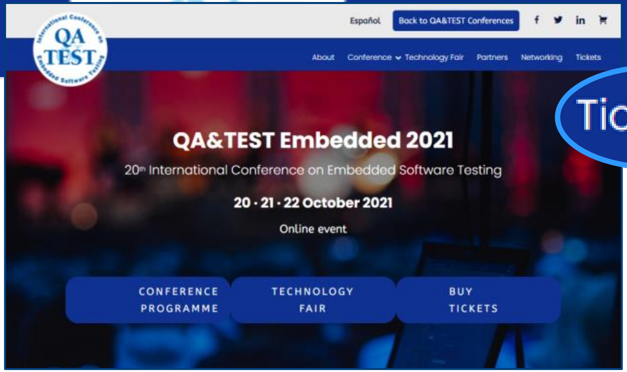
sdf423(@(#@x

Input-structure driven

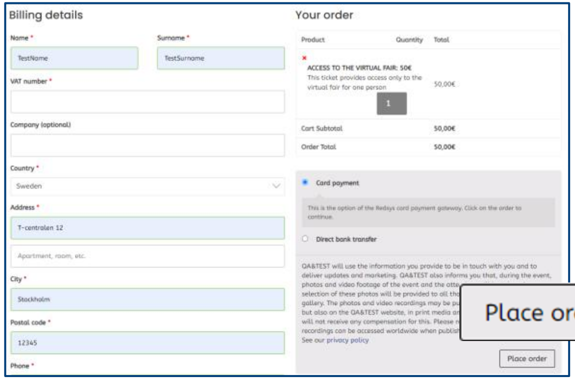


Input-structure driven





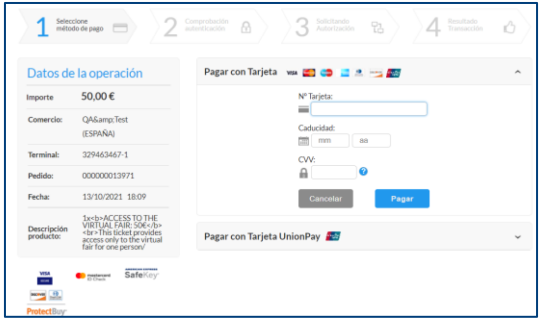
Tickets → Read cookie (if exists)



Place order

Save cookie

Register user



Program-structure driven

Types of Fuzzers

Input-seed driven	Input-structure driven	Program-structure driven
Random input generator	Input generator aware of types, field sizes, relation between fields	Generator aware of the program flow

Categories of Fuzzers

Input-seed driven	Input-structure driven	Program-structure driven
Generation based Inputs generated from scratch	**Dumb** Unaware of legitimate input structure	**White box** Fully aware of program structure
		Gray box Partially aware of program structure
Mutation based Inputs are based on previous inputs, coverage data, results	**Smart** Input structure aware knows how legitimate input looks like	**Black box** Unaware of program structure

Common Fuzzers

- Radamsa – mutation engine
- AFL/AFLplus – input seed/structure driven
- LibFuzzer – program structure driven
- HunggFuzz – input structure/program structure driven
- BooFuzz - input seed/structure driven
- Peach - input structure/program structure driven

Easier said…

“Generate” – How?

“Monitor results” – what’s expected?

A Diversion:

Security Mitigations

- Compiler flags
- Improve the run-time immunity to buffer overflows, out-of-array-bound errors, stack-based attacks etc.
- Examples:
  - Sanitizers:
    - Stack canary
    - ASAN
  - HW architecture / instruction set
    - CET
    - CFI

When triggered: Crash the program

Monitoring Fuzzing results

Question:
What's the expected result to each fuzz test case?

Answer:
In most cases: We don't know…

Monitoring Fuzzing results

Solution:

- Compile with security mitigation flags
- Run the fuzzer
- Crash = found potential bug!

Security mitigations’ role in Fuzzing

- Subtle bugs become deterministic crashes
- Reproduction is simple
- Mitigations can be used with any fuzzing tool
- Fuzzing without mitigations lose much of the fuzzing benefits

Example: Fuzzing Open Source code

- Code under test: **imgstats** utility, part of **imscript** (a collection of small and standalone utilities for image processing, written in C)
  https://github.com/mnhrdt/imscript

- Fuzzer: AFLplusplus
  https://github.com/AFLplusplus/AFLplusplus

- Makefile modified with:

```
CC = afl-gcc -fstack-protector-strong -fsanitize=address
```

Example: Fuzzing Open Source code

```
                  american fuzzy lop ++3.15a (default) [fast] {0}
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 2 days, 6 hrs, 13 min, 17 sec      │  cycles done : 55     │
│   last new path : 0 days, 11 hrs, 38 min, 42 sec     │  total paths : 747    │
│ last uniq crash : 0 days, 19 hrs, 54 min, 56 sec     │ uniq crashes : 35     │
│  last uniq hang : 1 days, 9 hrs, 18 min, 53 sec      │   uniq hangs : 16     │
├─ cycle progress ───────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 505.2071 (67.6%) │    map density : 0.01% / 0.03%         │
│ paths timed out : 0 (0.00%)        │ count coverage : 2.36 bits/tuple       │
├─ stage progress ───────────────────┼─ findings in depth ────────────────────┤
│  now trying : splice 15            │ favored paths : 294 (39.36%)           │
│ stage execs : 22/33 (66.67%)       │  new edges on : 367 (49.13%)           │
│ total execs : 17.0M                │ total crashes : 14.5k (35 unique)      │
│  exec speed : 0.00/sec (zzzz...)   │  total tmouts : 34.4k (247 unique)     │
├─ fuzzing strategy yields ──────────┴───────────┬─ path geometry ────────────┤
│   bit flips : disabled (default, enable with -D) │    levels : 25            │
│  byte flips : disabled (default, enable with -D) │   pending : 79            │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 0             │
│  known ints : disabled (default, enable with -D) │ own finds : 746           │
│  dictionary : n/a                                │  imported : 0             │
│havoc/splice : 548/6.00M, 233/10.9M               │ stability : 100.00%       │
│py/custom/rq : unused, unused, unused, unused     ├───────────────────────────┘
│    trim/eff : 26.11%/159k, disabled              │          [cpu000:112%]
└──────────────────────────────────────────────────┘
```

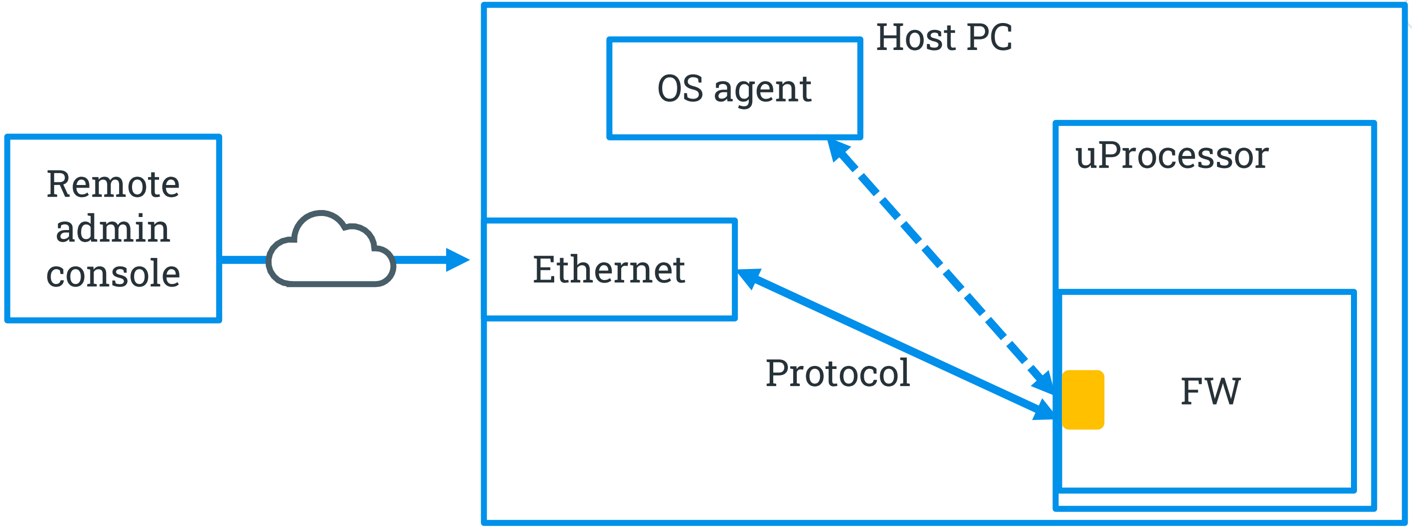
2.

DUT Overview

The Embedded System

- Internal FW running on an Intel uProcessor
- Connects to external entity (e.g. remote admin console; agent on the OS) to exchange information, and for configuration

The DUT



Goal:
Fuzzing of the protocol command processing in the FW code

The protocol

- Request-Response protocol
- In our system:
  - Requests generated by the FW
  - Responses from the admin console (or from the agent)

3.

Embedded system fuzzing challenges

Challenges

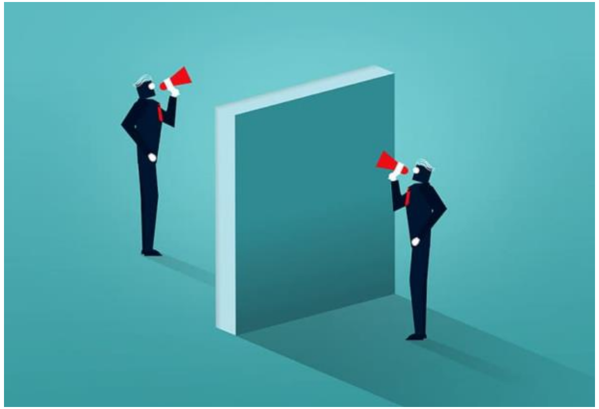
- Image size
- Synchronization with test machine
- Coverage feedback
- Crash detection
- Monitoring tools
- Target isolation

Challenges: Image Size

- Instrumenting a target code for a feedback/input based fuzzer increases the SW/FW image size significantly
- Example:
  - 800KB image w/o instrumentation
  - 1100KB after instrumentation

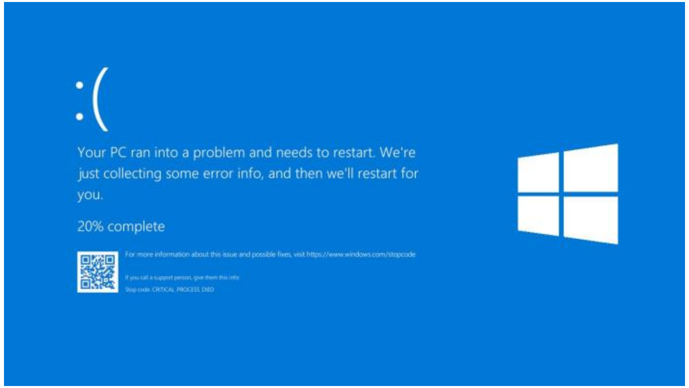
Challenges: Feedback path

- Smart fuzzers' mutation engines require code-coverage feedback
- No natural channels to pass the feedback to the fuzzer
  - Require innovative methods to pass the feedback
- Example:
  - In-system memory allocation for coverage information
  - Test hooks for pulling / pushing the information
- Side effect: Even larger memory requirements



Challenges: Crash Detection

- Most embedded system do not have a proper crash detection mechanisms (e.g. dump system, monitor, debugger)
- Prohibited by cost, code size considerations



Challenges: Monitoring Tools

- Embedded SW/FW programs lack standard monitoring tools (e.g. debugger, power monitors, perf etc.)
- Result: debugging and determination of system states is extremely difficult

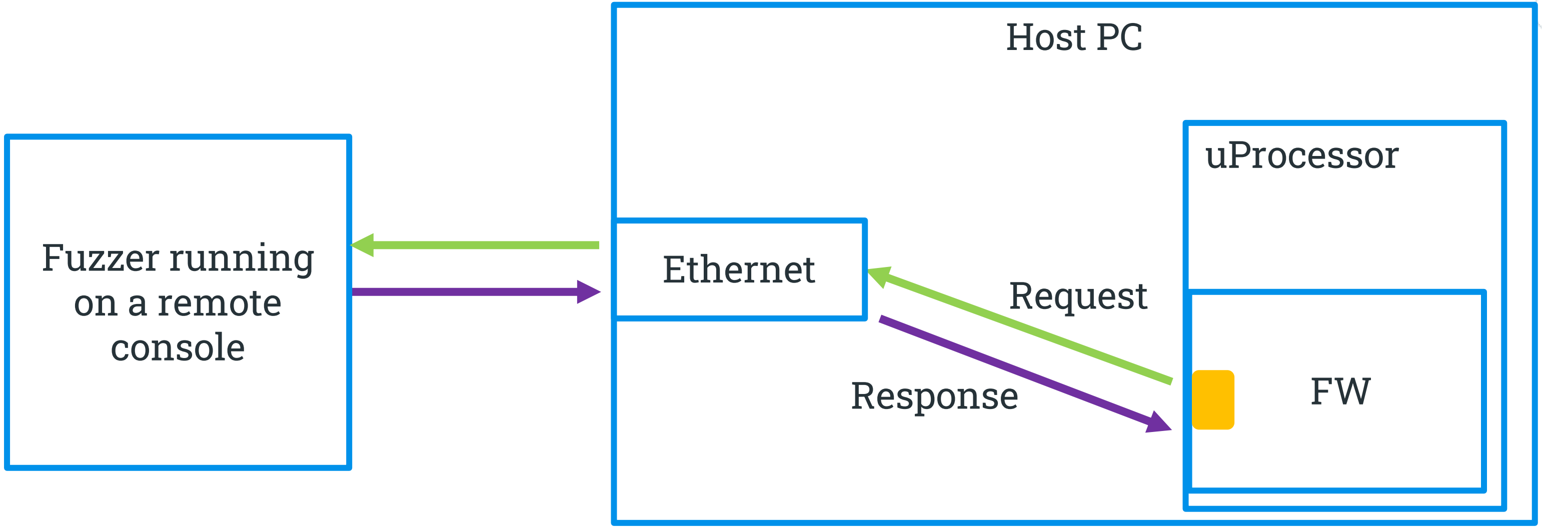
Challenges: Target Isolation

- A System of Systems challenge
- Isolating the target from the full system may be hard or impossible (e.g. Wi-Fi FW on IoT SoC)

4.

Fuzzer Architecture

The DUT


Fuzzer running on a remote console
Host PC
Ethernet
uProcessor
Request
Response
FW

Fuzzing in theory

- Wait for FW to send a request
- Identify the request
- Fuzz a response
- Send the fuzzed response
- Monitor the FW for hangs, crashes etc.

Problem:
- Inefficient
- Can't guarantee all requests → Not all responses are fuzzed

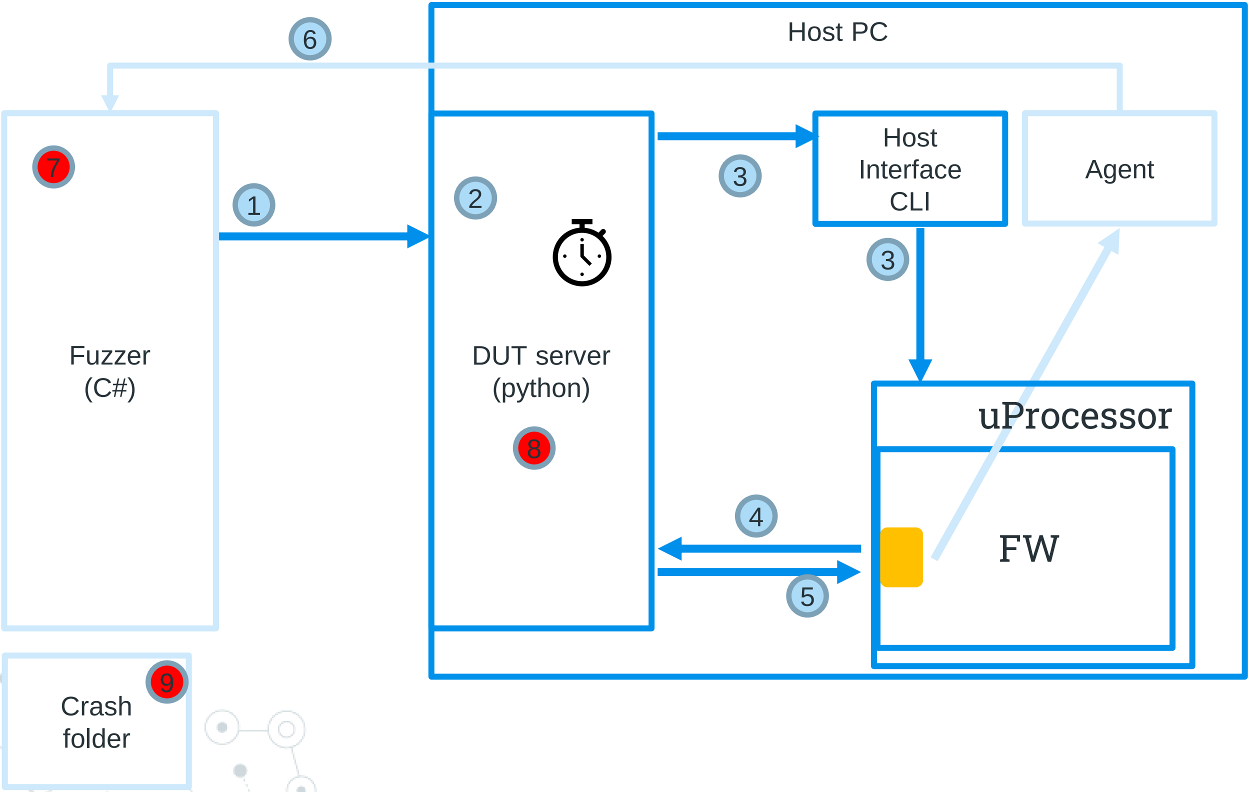
Actual Fuzzing Flow

- Randomly pick a response
- Fuzz the response data
- Use a test hook to trigger a request for the selected response
- Send the fuzzed response once the specific request arrives
- Sent feedback info via debug channel
- Monitor the FW for hangs, crashes, errors

Result:

- Efficient
- All requests generated; all responses fuzzed

Fuzzer architecture



1. The fuzzer creates a fuzzed response (25 to choose from)
2. The DUT server identifies the associated request
3. The DUT server triggers the request via Host Interface and test hook in the FW; Starts a time-out timer
4. The FW generates the request
5. The DUT server sends the fuzzed response
6. AFL code in the FW sends feedback to the fuzzer
7. Identify “crash “ feedback
8. If the next cycle fails (timeout), either this or previous cycle caused it
9. Save last 200 fuzzed commands to crash folder

5.

Results & Lessons-learned

Productization

- Fuzzer User Manual
  - Overview
  - Setup instructions
  - FW compilation instructions
  - First level debug and repro instructions
- All needed code, executables, pre-requisites on a shared folder or source repository

Results

- Fuzzer ran for two weeks
- Identified one ASAN failure
  - Good news / Bad news situation…
- Achieved confidence in the code’s robustness

Lessons Learned

- Fuzzing an embedded system – possible, but not trivial
- Feedback mechanism must be designed and implemented
- May call for test hooks
- Compilation with sanitizers: limit to the code-under-test
  - Reduce binary size to the minimum needed
  - Can be controlled by CMAKE scripts
- Do proper documentation to avoid losing the capability
- ROI: difficult to assess
  - How often / how long to run the fuzzer?
  - What's the worth of "removed vulnerability"?
  - As Secure Code Development improves, fuzzing may yield less results

Thanks!

Any questions?

You can find us at:

dor.levy@intel.com

michael.m.stahl@gmail.com / www.testprincipia.com