Efficient and Effective Fuzz Testing of Automotive Linux Systems using Agent Instrumentation

Dennis Kengo Oka and Rikke Kuipers

dennis.kengo.oka@synopsys.com
rikke.kuipers@synopsys.com

Automotive Linux Summit, Tokyo, Japan
2019/7/19

Dennis Kengo Oka - Automotive Security

- Started Automotive Security in 2006
  - Securing over-the-air updates and remote diagnostics
- Senior Solution Architect based in Tokyo but involved in automotive security globally
- Long experience working with and supporting several OEMs and suppliers on improving their security processes and practices
- Member of Jaspar (Japan Automotive Software Platform and Architecture) security working group
  - Participating in standardization/best practices work for automotive industry in Japan
  - Contributed to writing “Bluetooth Fuzzing Guideline” shared with OEMs/suppliers in Japan
- 60+ publications and presentations at e.g. escar, JSAE, SAE World Congress, Code Blue, IEEE Cybersecurity etc.

Challenges of fuzzing automotive components

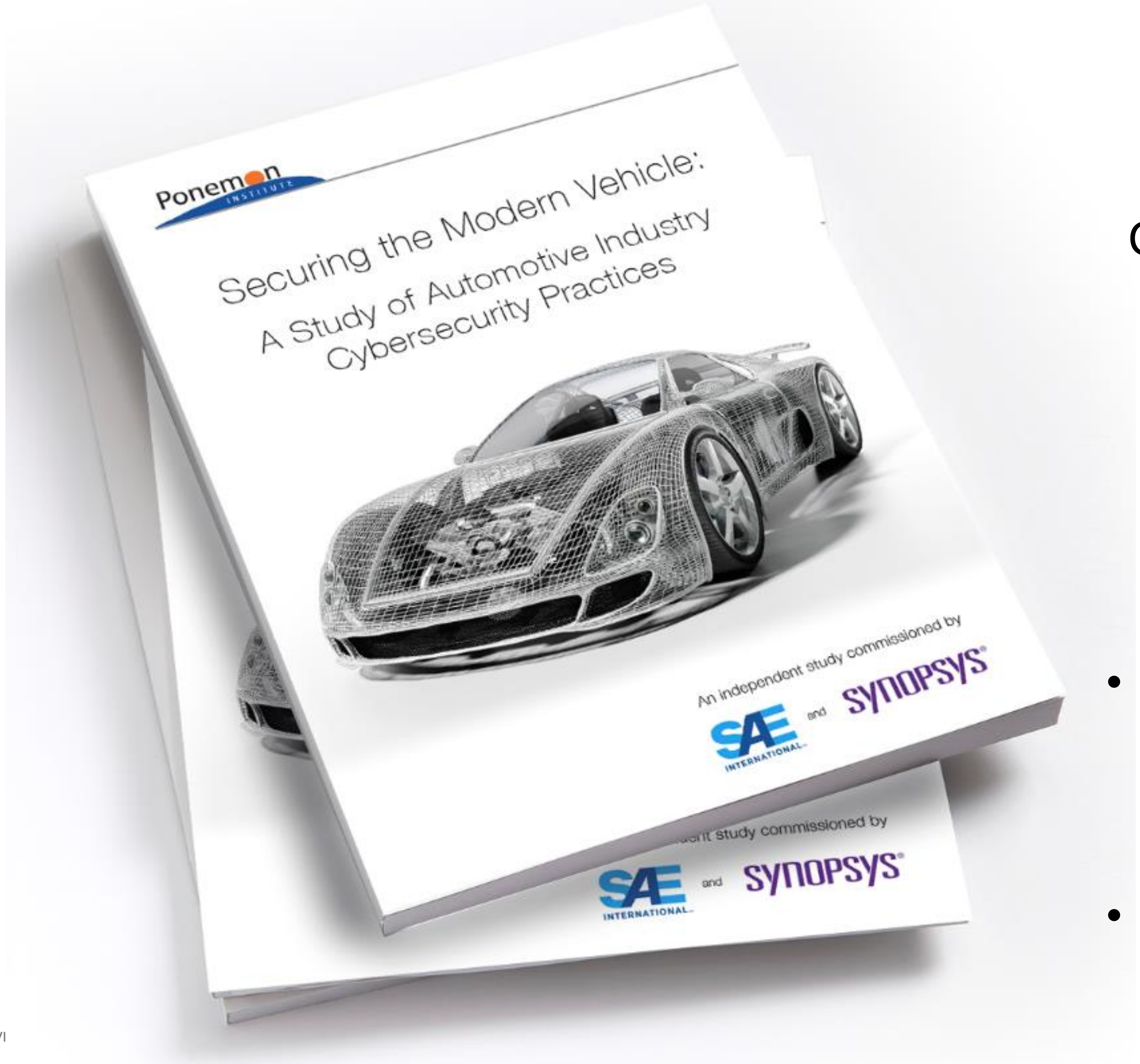
Agent Instrumentation Framework - using agents on the SUT to improve fuzz testing

Results show it is possible to detect previously undetectable exceptions

Challenges of fuzzing automotive components

Agent Instrumentation Framework - using agents on the SUT to improve fuzz testing

Results show it is possible to detect previously undetectable exceptions



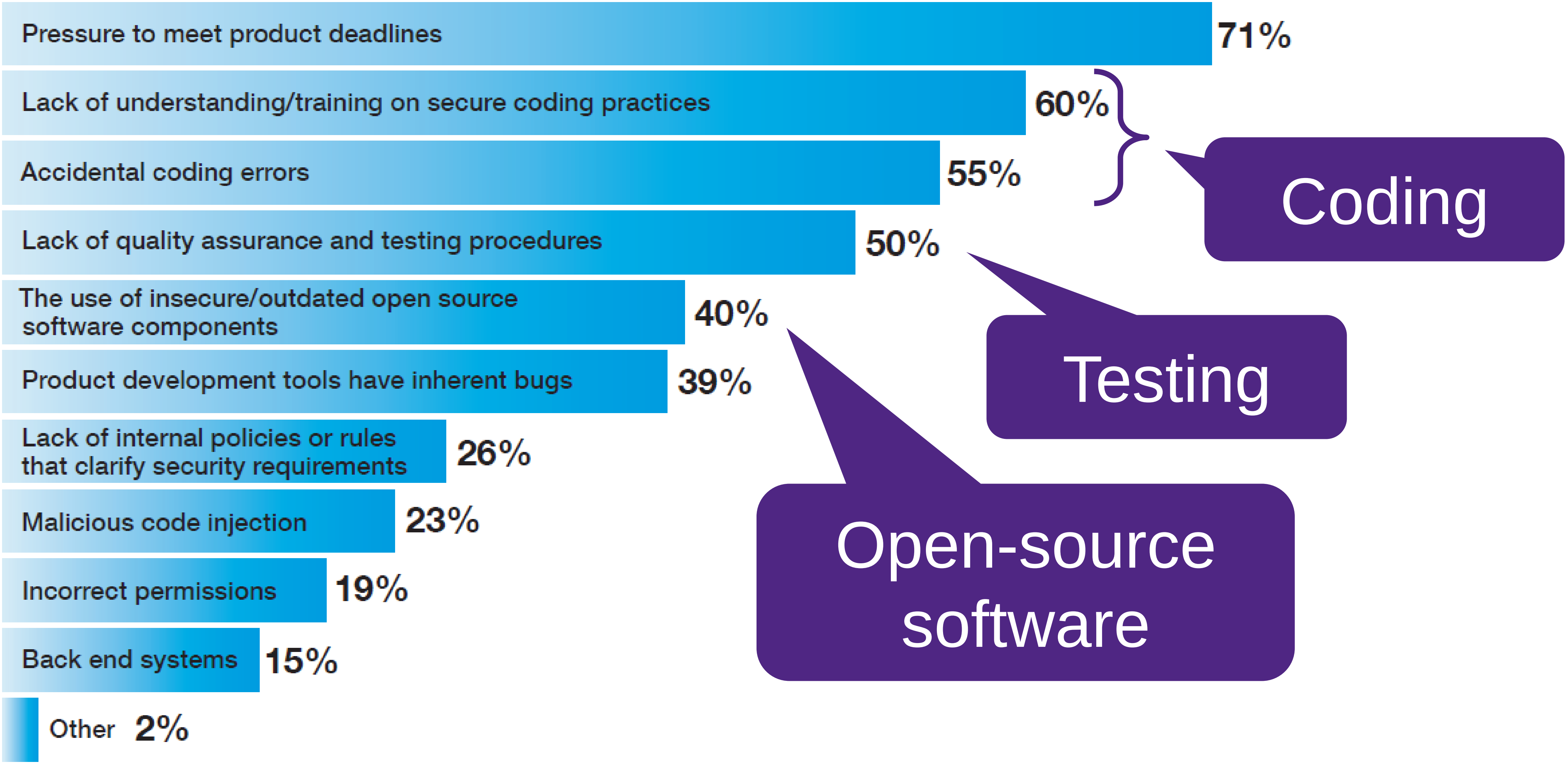
Download Link:
www.synopsys.com/auto-security

Cybersecurity Research Center

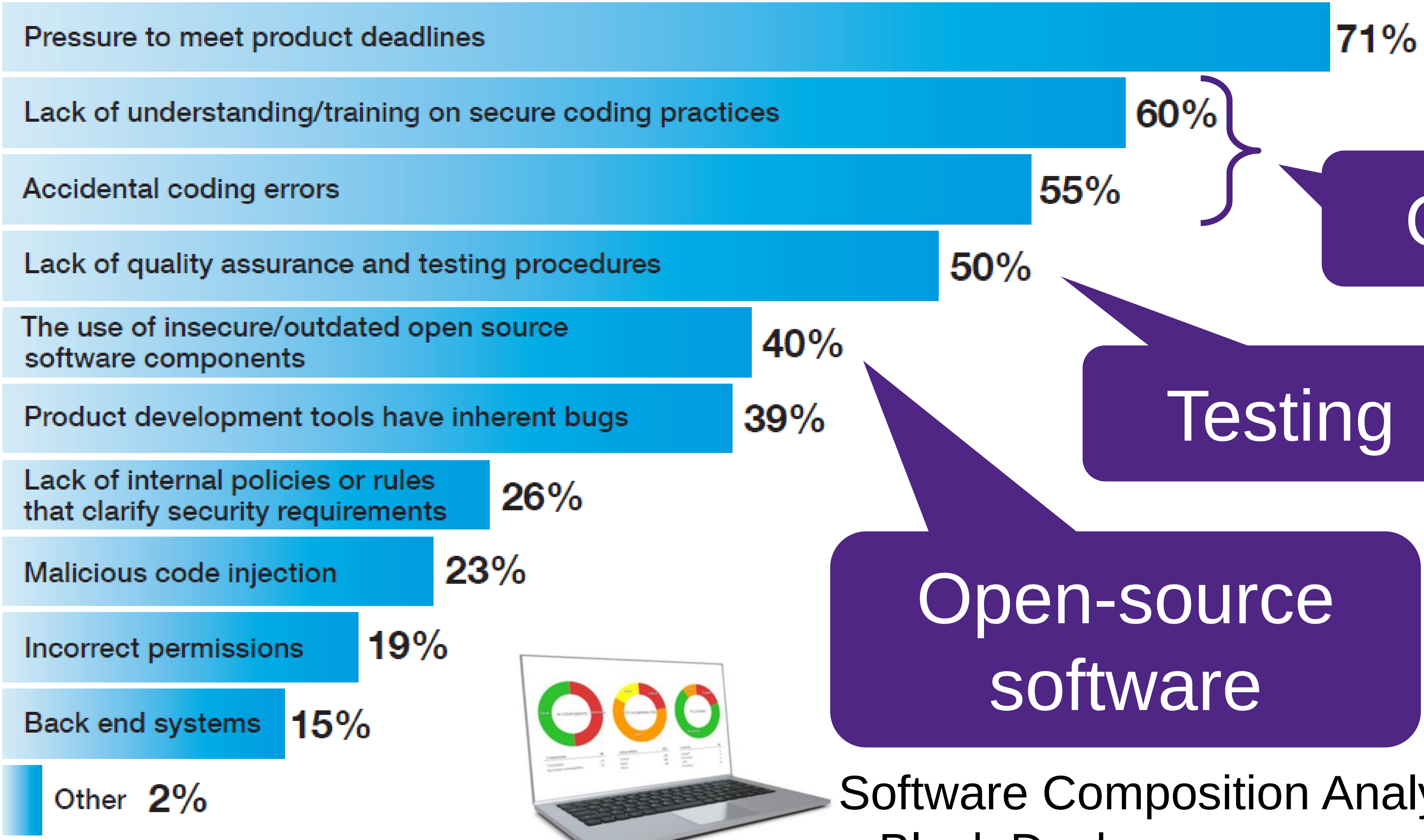
CyRC

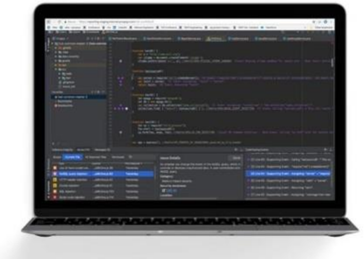
- Securing the Modern Vehicle: A Study of Automotive Industry Cybersecurity Practices
- 2019 Open Source Security and Risk Analysis (OSSRA) Report

What are the primary factors that lead to vulnerabilities in automotive software/technology/components?



What are the primary factors that lead to vulnerabilitie automotive software/technology/components?

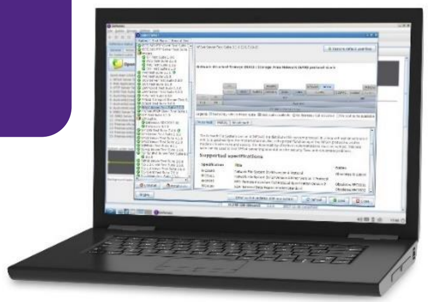




Static Analysis – Coverity

Coding

Testing



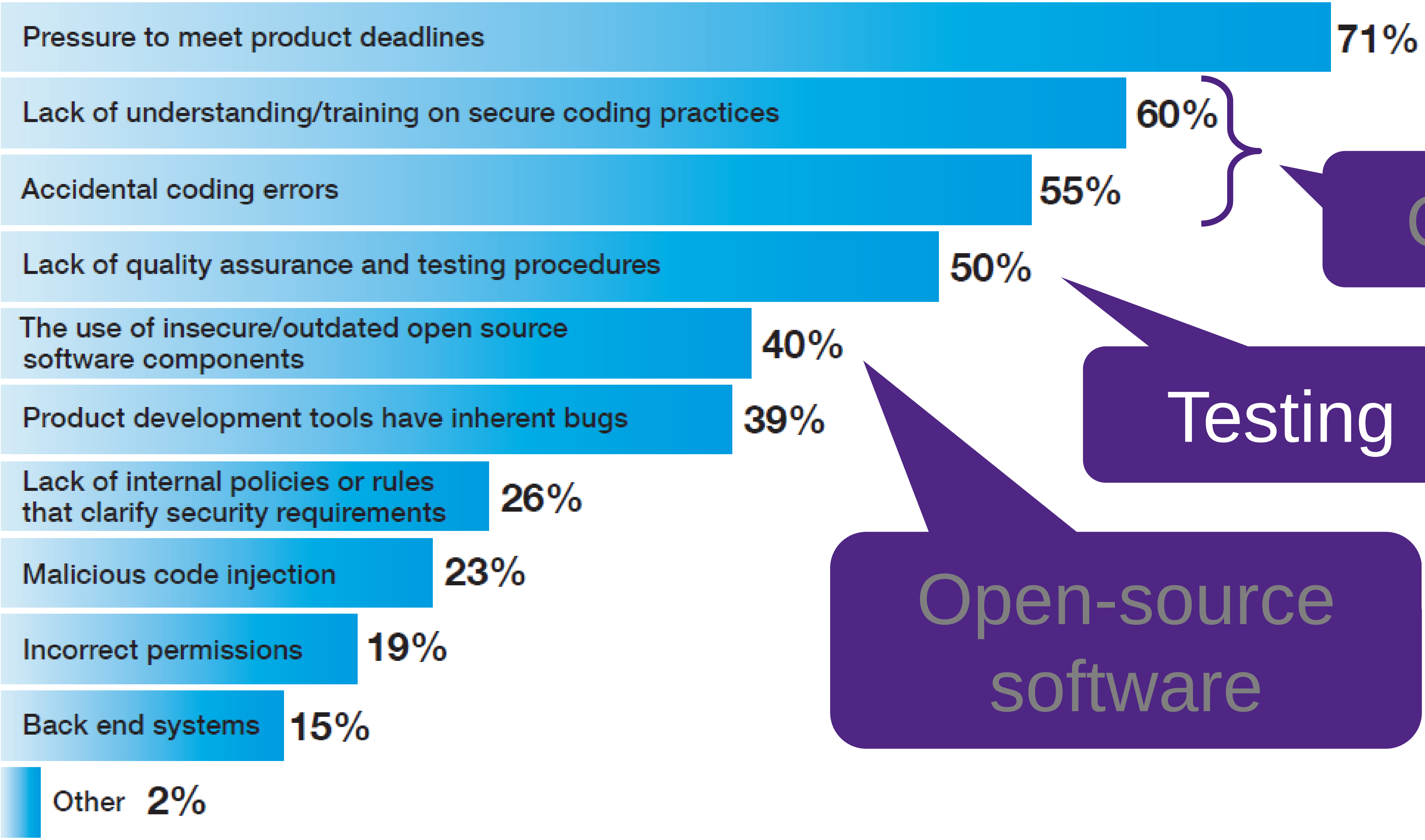
Open-source software

Fuzz Testing – Defensics

Software Composition Analysis – Black Duck

SYNOPSYS®

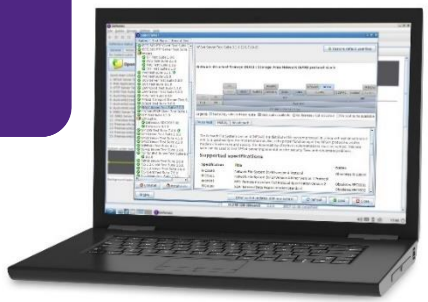
What are the primary factors that lead to vulnerabilities in automotive software/technology/components?



Coding

Testing

Open-source software



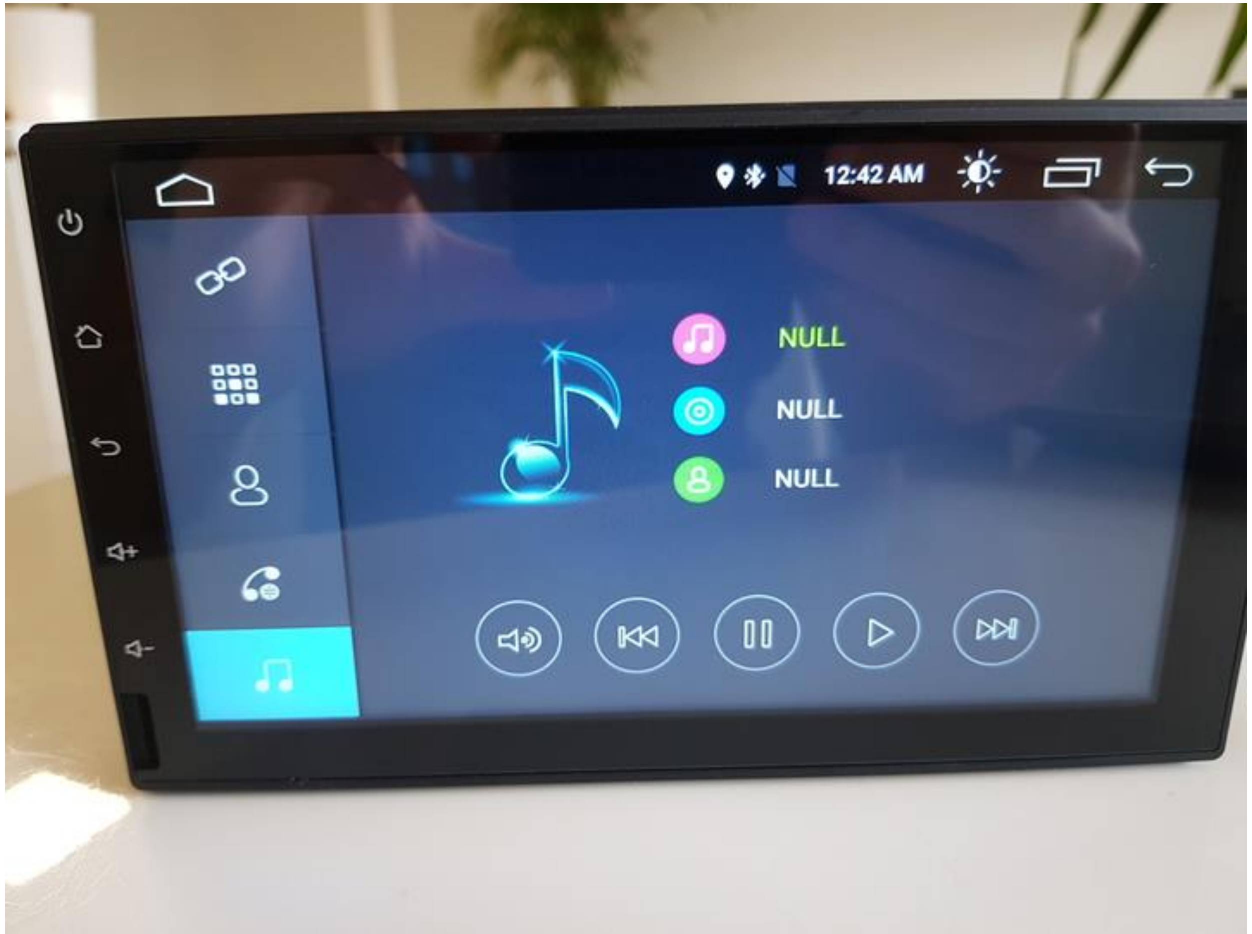
Fuzz Testing – Defensics

SYNOPSYS®

Fuzz Testing

- Testing technique where malformed or "out-of-specification" inputs are provided to the SUT (system under test) which is then observed to detect exceptions or unintended behavior

- Allows developers and testers to identify unknown vulnerabilities in their systems and components



Critical Vulnerabilities Detected by Fuzz Testing

- ASN.1/SNMP various vulnerabilities (2001/2002)
- Apache IPv6-URI vulnerability (2004)
- Image file format various vulnerabilities (2005)
- XML library various vulnerabilities (2009)
- Linux Kernel IPv4 and SCTP several vulnerabilities (2010)
- RSA signature verification vulnerability in strongSwan (2012)
- **Heartbleed: OpenSSL vulnerability (2014)**
- OpenSSL and GnuTLS several vulnerabilities (2004, 2008, 2012, 2014)
- **Badlock: Samba/DCE-RPC denial-of-service vulnerability (2015)**

Real World Example – Two Embedded Systems

Low level ECU
Powertrain, chassis, body

Small embedded code base
Model-based development

Safety criticality: High/Low
Security exposure: Low

IVI (In-vehicle infotainment), ADAS,
Telematics ECU

Open source platforms/libraries
Entire operating systems (AGL)

Safety criticality: Low/Medium/High
Security exposure: High

Challenges for Automotive Components

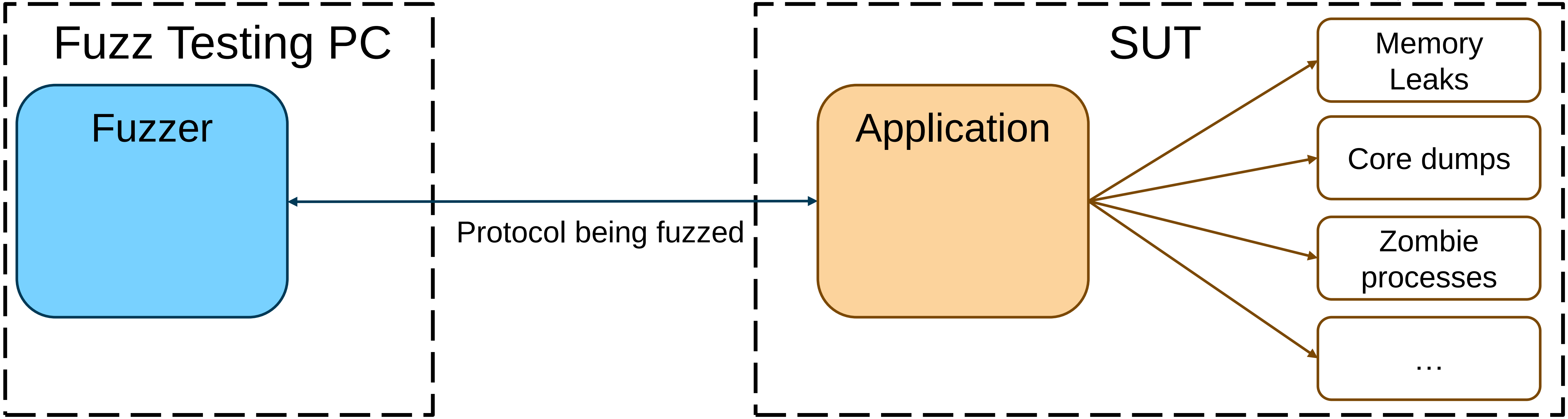
- Difficult to instrument the SUT in a proper way to determine whether there was an exception or if it has failed or crashed

- E.g., fuzz testing Wi-Fi or Bluetooth, instrumentation typically occurs over the same protocol that is being fuzzed – this limited instrumentation could lead to several potential issues going undetected

- Difficult to gather information from the SUT to be able to easier determine the underlying root causes for the exception or failure

Undetected Exceptions on SUT

Exceptions on SUT not detected by just observing the fuzzed protocol



Challenges of fuzzing automotive components

Agent Instrumentation Framework - using agents on the SUT to improve fuzz testing

Results show it is possible to detect previously undetectable exceptions

Agent Instrumentation Framework - Concept

- Developers/testers have access to internals of the SUT (e.g., Linux and Android)
- Achieve more efficient and accurate fuzz testing by employing a gray/white box approach where Agents placed on the SUT assist with the instrumentation
- The Agents provide the fuzzer with detailed instrumentation data from the SUT
- This data is used to determine the fail / pass verdict of a test case and also provides the developers/testers with valuable information from the SUT

Agent Instrumentation Framework - Modes

Synchronous mode

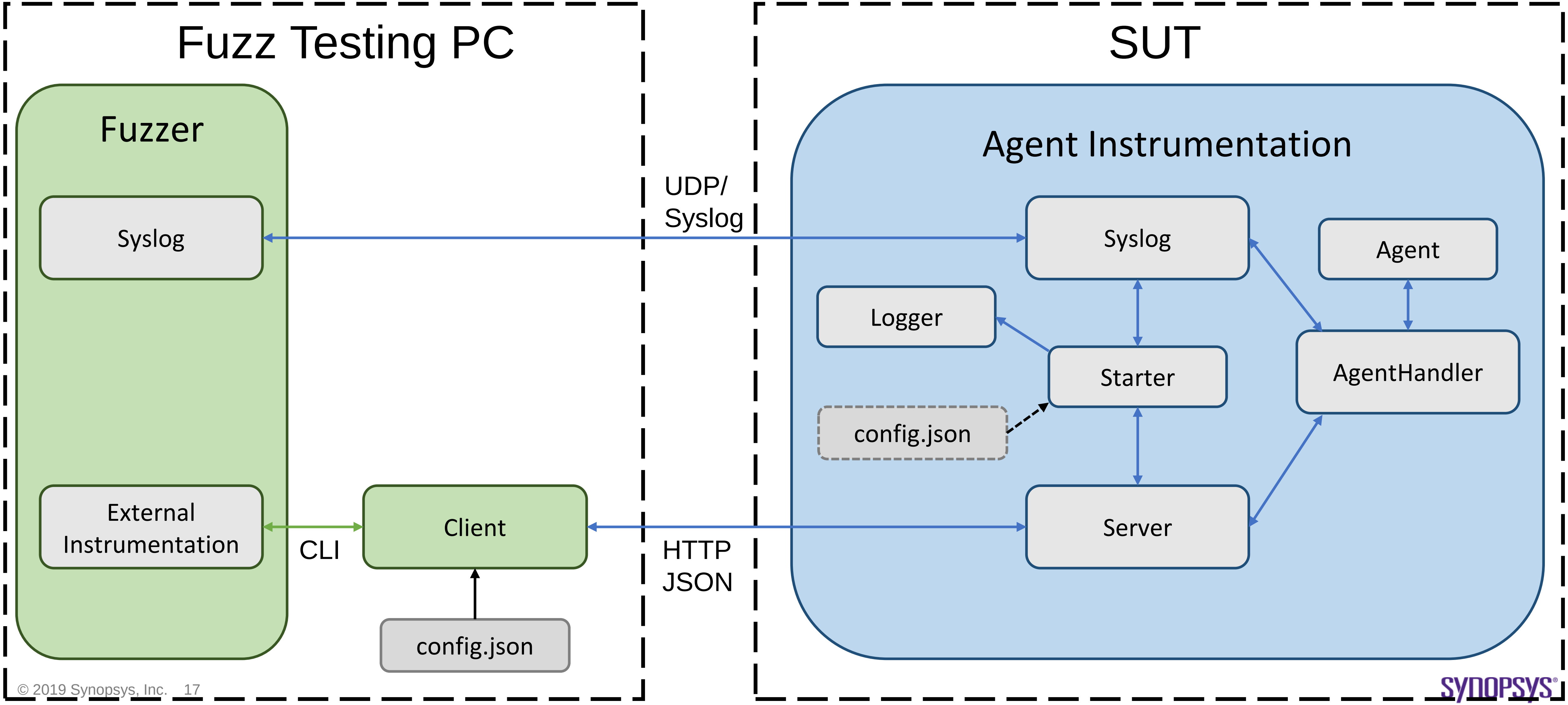
- More control over the Agents
- Agents can perform various functions before and after a test case executes
- Allows for automation and using more advanced techniques for finding unknown vulnerabilities
- Slower to execute the test run

Asynchronous mode

- Agents are polled periodically
- Fuzzer parses incoming syslog messages with instrumentation data
- Cannot tie an exception to a specific test case but only reports when a prespecified condition has been met
- Advantage is speed of execution with the loss of accuracy

SYNOPSYS®

Overview of Agent Instrumentation Framework



Examples of Agents (1)

- AgentCoreDump
  - Looks for a core dump file, if detected will give a fail verdict
  - This file can be used to further analyze the state of the process during the crash
- AgentLogTailer
  - Monitors a log file
  - If a new line is written to the file and matches any of the predefined parameters, it will give a fail verdict
- AgentProcessMonitor
  - Monitors the state of a process
  - Gives a fail verdict if the process is down or turned into a zombie process
  - Can also monitor the process' memory usage and give a fail verdict if the usage goes over a configured limit

Examples of Agents (2)

- AgentPID
  - Monitor processes on the SUT with more options
  - Can only be run in synchronous mode, otherwise it might generate false positives
  - Before each test case is executed, a mapping is made of each predefined process with its process identifier (PID) and PIDs of its children
  - After a test case is executed the same mapping is performed again
  - If a process has died its PID will not be present in the new mapping
  - If a process has died but restarted then it has new PID
  - A fail verdict is issued in both cases

Examples of Agents (3)

- AgentAddressSanitizer
  - Finds memory addressability issues and memory leaks using Google's ASAN framework:
    - e.g., Use after free (dangling pointer dereference), Heap buffer overflow, Stack buffer overflow, Global buffer overflow, Use after return, Use after scope, Initialization order bugs, Memory leaks
  - Target software needs to be compiled with additional compiler flags, can only be run in synchronous mode
  - Two configurations: memory leaks and all other addressability issues
  - To find memory leaks: target process is killed after each testcase and ASAN's output is analyzed
  - To find all other addressability issues: the Agent configures ASAN to kill the process upon finding any issue – this configuration is faster than finding memory leaks
  - In both cases, crash trace and if available detailed crash information are reported back to the fuzzer

Examples of Agents (4)

- AgentValgrind
  - Finds memory leaks and addressability issues (similar to AgentAddressSanitizer)
  - Difference is the mode of operation and speed
  - Uses various checkers and profilers from the Valgrind project, which effectively emulates a hardware layer for the program to run on
  - Quite heavy and adds overhead to each test case
  - Advantage is that there is no need to recompile target software

Example Configuration Script

```
{
	"instrumentation_method": "external",
	"external": {
		"ip": "192.168.0.2",
		"port": 8881,
		"token": "Secret"
	},
	"before_run": {
		"agents": {
			"pid_monitor": {
			"type": "AgentPID",
			"executables": [
				"wpa_supplicant",
				"bluetoothd",
				"lightmediascannerd",
				"geoclue"
			]
			}
		}
	}
}
```

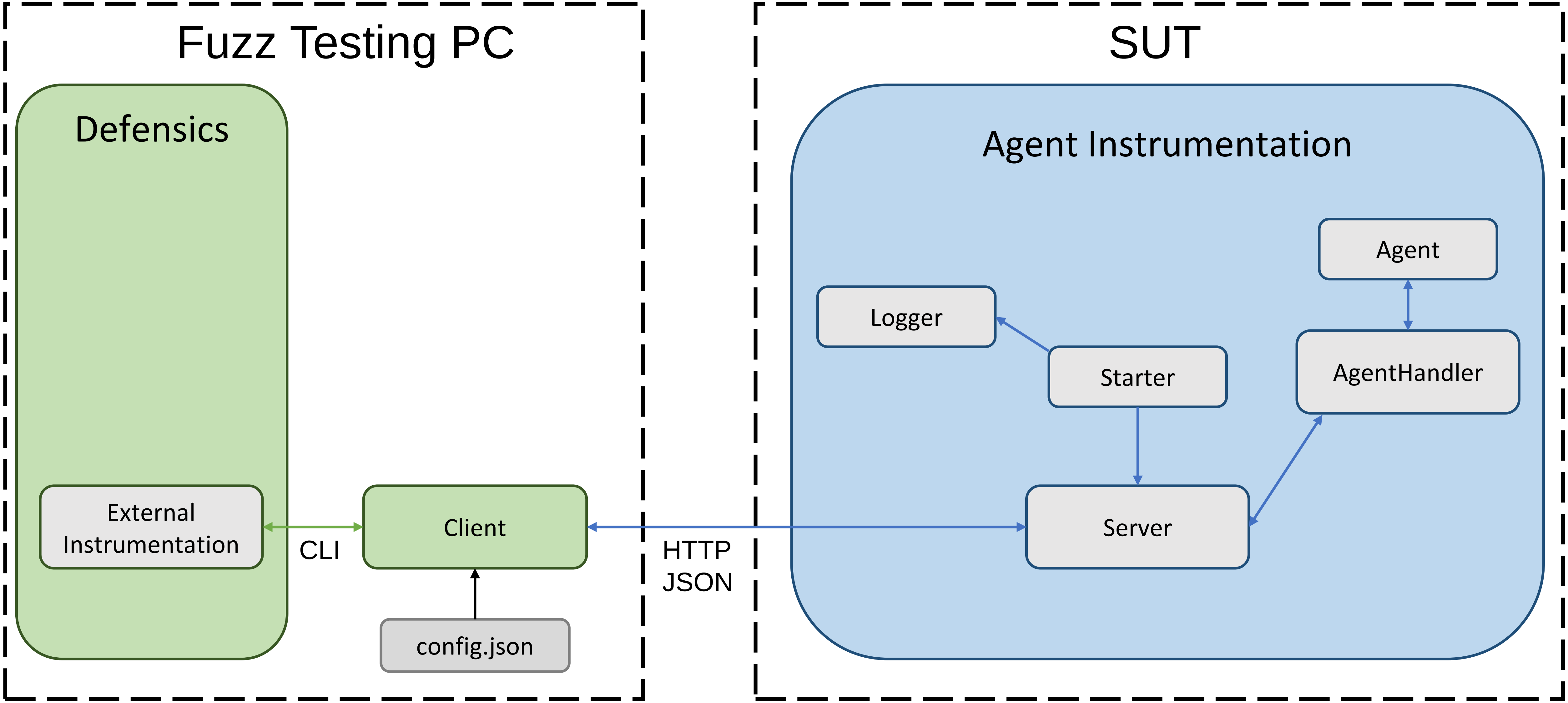
Starter script on SUT loads AgentPID which monitors the specified processes

Challenges of fuzzing automotive components

Agent Instrumentation Framework - using agents on the SUT to improve fuzz testing

Results show it is possible to detect previously undetectable exceptions

Setup of Test Bench



Test Setup

- Run in synchronous mode
- Agents used most:
  - AgentPID as it requires no extra configuration or modification of the system under test
  - AgentAddressSanitizer and AgentValgrind to find memory issues
- Automatically run scripts to enable the agents and collect results

- Target systems: Linux or Android
- Protocols:
  - Browser protocols
  - Bluetooth protocols
  - Wireless (802.11) protocols
  - Messaging protocols
  - File format parsing (audio, images, video)
  - CAN-bus

Bluetooth Results

AgentPID agent

- Found a critical vulnerability where a fuzzed single frame causes the main Bluetooth kernel module to crash
- Detected by monitoring the bluetoothd daemon process using the AgentPID agent
- Since the device has a daemon watchdog which quickly restarts bluetoothd, this vulnerability would not have been detected by just observing the Bluetooth protocol

15:25:43.238 python client.py --config pid-monitor.json instrumentation

15:25:43.640 Instrumentation verdict: FAIL

15:25:43.640 FAIL Agent: pid_monitor Info: Agent pid_monitor says

15:25:43.640 ofonod : ['353']

15:25:43.640 bluetoothd : ['896'] -> [].

15:25:43.640 bluetoothgateway : [547]

15:25:43.640 mediaserver : [154]

15:25:43.640 wez-launch : ['114']

15:25:43.640 wez : ['114', '130']

15:25:43.640 ogg_streamhandler : [11]

15:25:43.640 pulseaudio : ['775']

15:25:43.640 audio_daemon : ['839']

15:25:43.640 media_engine_app : [145]

SYNOPSYS®

Wi-Fi Results

Log Tailer agent

- Found a critical vulnerability where a fuzzed single frame caused several kernel modules to crash
- Non-authenticated frame – could be sent by anyone
- Log Tailer agent found this issue by tailing syslog with the keywords “stack”, “crash” and several kernel module names
- This vulnerability would not have been detected by only observing the Wi-Fi communication as the kernel watchdog restarted the affected modules immediately

```
[  +0.000032] ------------[ cut here ]------------
[  +0.000019] WARNING: CPU: 3 PID: 912 at drivers/net/wireless/[redacted]
[  +0.000002] Modules linked in: loop(O) [redacted]
[  +0.000083] CPU: 3 PID: 912 Comm: [redacted] Tainted: G     U  W
[  +0.008363] task: edfbab40 task.stack: ecf66000
[  +0.005063] EIP: iwl_mvm_tx_mpdu+0x1a7/0x3d7 [iwlmvm]
[  +0.000003] EFLAGS: 00010286 CPU: 3
[  +0.000002] EAX: 0000001f EBX: ee75cde4 ECX: f4670344 EDX: f466ab4c
[  +0.000002] ESI: 00000002 EDI: 000001a0 EBP: ecf67bdc ESP: ecf67ba0
[  +0.000003]  DS: 007b ES: 007b FS: 00d8 GS: 0000 SS: 0068
[  +0.000002] CR0: 80050033 CR2: a63de000 CR3: 2bcd8ec0 CR4: 001006f0
[  +0.000002] Call Trace:
[  +0.002741]  iwl_mvm_tx_skb+0x5b/0x139 [iwlmvm]
[  +0.005071]  iwl_mvm_mac_tx+0x9c/0x144 [iwlmvm]
[  +0.005068]  ? iwl_mvm_stop_ap_ibss+0x12e/0x12e [iwlmvm]
[  +0.005952]  ieee80211_tx_frags+0x17b/0x192 [mac80211]
...
```


SYNOPSYS®

MQTT Results

- Found a memory leak in the popular MQTT broker Mosquitto
- Access to the source code of the MQTT broker
- AgentAddressSanitizer to test for memory addressability issues and memory leaks
- Recompiled the code with additional compilation flags to enable Google's ASAN instrumentation
- This vulnerability would not have been detected by only observing the fuzzed protocol

AgentAddressSanitizer

```
21:34:37 TEST CASE #29
21:34:37 mqtt.connect-disconnect.connect.element: Underflow of 12 -10 =2 oc
21:34:37 tcp 45264 --> localhost:1883 4 MQTT CONNECT ANOMALY!
21:34:37 Receiving connack over tcp failed: expected (0b0010) but got ()
21:34:37 Instrumenting (1. round)...
21:34:37 /usr/bin/python2 /home/p0c/synopsys/aif/client.py      --config /h
21:34:37 Instrumentation verdict: FAIL
21:34:37 FAIL Agent: memory_mqtt Info: Agent memory_mqtt says Memory leak f
21:34:37
21:34:37 =================================================================
21:34:37 ==20==ERROR: LeakSanitizer: detected memory leaks
21:34:37
21:34:37 Direct leak of 1 byte(s) in 1 object(s) allocated from:
21:34:37     #0 0x7f6af581ed99 in __interceptor_malloc /build/gcc/src/gcc/l
21:34:37     #1 0x56218adca9e3 in _mosquitto_malloc (/home/p0c/mosquitto/sr
21:34:37     #2 0x56218ade0802 in _mosquitto_read_string (/home/p0c/mosquit
21:34:37     #3 0x56218ade5d85 in mqtt3_handle_connect (/home/p0c/mosquitto
21:34:37     #4 0x56218ade2e77 in mqtt3_packet_handle (/home/p0c/mosquitto/
21:34:37     #5 0x56218ade2b61 in _mosquitto_packet_read (/home/p0c/mosquit
21:34:37     #6 0x56218adca6b7 in loop_handle_reads_writes (/home/p0c/mosqu
21:34:37     #7 0x56218adc891c in mosquitto_main_loop (/home/p0c/mosquitto/
21:34:37     #8 0x56218ada185c in main (/home/p0c/mosquitto/src/mosquitto+0
21:34:37     #9 0x7f6af430606a in __libc_start_main (/usr/lib/libc.so.6+0x2
21:34:37
21:34:37 SUMMARY: AddressSanitizer: 1 byte(s) leaked in 1 allocation(s).
21:34:37
```


SYNOPSYS®

Key Takeaways

Shift Left: Conduct Fuzz Testing earlier in the software development lifecycle

- Allows for gray/white box approach
- Finding issues earlier
- Fixing issues earlier!

Use Agent Instrumentation for efficient and effective Fuzz Testing

- Advanced instrumentation to detect exceptions and unintended behavior not observable over the fuzzed protocol
- Collect additional information from the SUT to help developers identify the root cause
- Allows for automating fuzz testing

Dennis Kengo Oka
dennis.kengo.oka@synopsys.com

- Challenges of fuzzing automotive components
- Agent Instrumentation Framework - using agents on the SUT to improve fuzz testing
- Results show it is possible to detect previously undetectable exceptions