SYNOPSYS®

WHITE PAPER

# Improving Fuzz Testing of Infotainment Systems and Telematics Units Using Agent Instrumentation

By Rikke Kuipers and Dennis Kengo Oka, Synopsys

# Table of contents

# Overview

In the past few years, cyber security has become more intertwined into each step of the automotive development process. In particular, fuzz testing has proven to be a powerful approach to detect unknown vulnerabilities in automotive systems. However, with limited instrumentation, especially on systems such as in-vehicle infotainment (IVI) systems and telematics units, several types of issues go undetected, such as memory leaks and cases where the application crashes but restarts quickly.

These systems are typically based on operating systems that provide more functionality, such as Linux and Android. Therefore, it is possible to use appropriate tools to collect additional information from the system under test (SUT) to determine whether any exceptions were detected during fuzz testing. Furthermore, it is possible to gather more details about the detected exceptions on the SUT, which helps developers to better understand and identify the root cause of the issues and fix the problems more efficiently.

To this end, we introduce the Agent Instrumentation Framework and explain how it can be used to improve fuzz testing of IVIs and telematics units. We show how additional information can be collected from the target system and used to identify any exceptions on the SUT to help developers identify the underlying cause of any issues detected. Finally, to showcase the effectiveness of the Agent Instrumentation Framework, we built a test bench based on this approach and performed fuzz testing on multiple SUTs. Based on our findings, we highlight several examples of issues that would not have been detected without agent instrumentation.

# Introduction

In recent years, there has been an increase of cyber security awareness in the automotive industry. The well-known Jeep hack presented in 2015 at Black Hat / DEF CON has played a major role in increasing this awareness. Multiple activities have improved the security stance in the industry, with various work published by NHSTA and VDA and standardization activities such as SAE J3061 Cybersecurity Guidebook for Cyber-Physical Vehicle Systems and ISO 21434 Road Vehicles—Cybersecurity engineering.

In the typical automotive development process, usually known as the V-model, cyber security has become an integral part of every step. Security activities include defining and reviewing security requirements; conducting security reviews of the design; applying best practices and secure coding standards during software development; using automated tools such as static code analysis tools; and performing testing activities such as security testing, fuzz testing, and penetration testing.[1] Specifically, regarding automotive security testing, "Security Crash Test—Practical Security Evaluations of Automotive Onboard IT Components" (Bayer, Enderle, Oka, and Wolf) provides a comprehensive overview of various testing methodologies that can be applied.[2] One extremely powerful testing approach is fuzz testing, which allows developers and testers to identify unknown vulnerabilities in their systems and components. Fuzz testing is a type of test approach where malformed or "out-of-specification" inputs are provided to the system under test (SUT), which is then observed to detect any exceptions or unintended behavior. Currently, there is rapid software development in the areas of connected cars and autonomous driving. These are both very software-heavy solutions, and since they interact with the externals of the vehicle, they are very attractive targets to attackers, as well as more prone to erroneous or malformed input coming from systems or the environment that is out of the control of the vehicle manufacturer or system manufacturer. Therefore, it is extremely important to perform fuzz testing of these types of systems to ensure security, robustness, and safety.

Several automated tools can be used to efficiently perform fuzz testing of automotive systems, in the sense that they support the relevant protocols over which to communicate with the SUT. However, one challenge with fuzz testing of automotive components is that it is often difficult to instrument the SUT in such a way as to determine whether there was an exception on the SUT or whether the SUT failed and crashed. Moreover, since testing is typically performed as a black box, it is difficult to gather enough information from the SUT to easily determine the underlying root causes for the exception or failure. This information would be invaluable in helping developers of the system find and fix the discovered issues more quickly and accurately.

Previous research has shown that it is possible to perform efficient fuzz testing of deeply embedded ECUs by instrumentation using HIL (hardware-in-the-loop) systems.[3] The solution presented in that research describes how to monitor the behavior of the ECU under test by measuring the analog and digital signals generated by the ECU in the HIL system. This integrated solution provides various ways for the HIL system to determine whether there is an exception on the SUT and can detect exceptions that would be undetectable if one were observing only the protocol being fuzzed. For example, if the CAN bus is being fuzzed, the target ECU may misbehave and generate analog or digital output to erroneously try to control an actuator. This unintended behavior would be missed if only the CAN bus were being observed. By instrumenting the ECU with a HIL system, it is possible to detect this abnormal behavior.

By contrast, in this paper, we focus on fuzz testing of the connected car, specifically in-vehicle infotainment (IVI) systems and telematics units. We present the concept of the Agent Instrumentation Framework, which can be used to better instrument these systems to allow for more efficient and accurate fuzz testing. That is, for these types of systems, additional approaches can be taken to improve instrumentation. Since these systems are typically based on operating systems providing more functionality, such as Linux and Android,[4] using the appropriate tools, it is possible to collect information from the SUT to determine whether any exceptions were detected during fuzz testing. Additionally, more details about the detected exceptions can be fed back to the fuzz testing tool and stored in the log file. This additional information helps developers better understand and identify the root cause and eventually fix the problem.

In this paper:

- We introduce the Agent Instrumentation Framework and explain how it can be used to improve fuzz testing of IVIs and telematics units.
- We show how additional information can be collected on the target system and used to determine whether there are exceptions and help developers identify the underlying cause of any issues detected.
- We built a test bench based on this approach and fuzz tested several SUTs. We highlight some examples of our findings that would not have been detected without agent instrumentation.

## Background and problem statement

Many automotive organizations have made fuzz testing a mandatory step in their software development process or are moving toward doing so. In other industries, such as network and telecommunications or enterprise, fuzz testing has already been integrated into the software development process and proven to be an effective approach to identifying bugs and vulnerabilities quickly. These target systems are typically easy to instrument by monitoring just the same protocol that is being fuzzed. This approach is common when fuzzing IT solutions such as a web server or a specific communication library.

In many cases, monitoring the same protocol that is being fuzzed is effective. For example, monitoring HTTP requests and corresponding HTTP responses could help identify potential unknown vulnerabilities in a web server. Furthermore, the famous Heartbleed vulnerability (CVE-2014-0160), which was found by fuzzing the OpenSSL library, was identified by monitoring the responses to the heartbeat request message.[5] By contrast, automotive systems are often more complicated and interconnected with other systems and therefore not easy to instrument. As a result, without proper instrumentation, many unknown vulnerabilities and potential issues cannot be identified on these automotive systems.

As shown in previous research,[6] without proper instrumentation of deeply embedded ECUs using HIL systems, several potential issues go undetected. Similarly, without proper instrumentation of IVIs and telematics units, numerous potential issues go undetected.

Typically, when performing fuzz testing of a specific protocol, such as fuzz testing over Wi-Fi or Bluetooth on an IVI, instrumentation occurs over the same protocol being fuzzed. In other words, the SUT is being monitored only over the same protocol being fuzzed. This limited instrumentation could result in several potential issues going undetected. We illustrate this in Figure 1 and describe in more detail as follows.

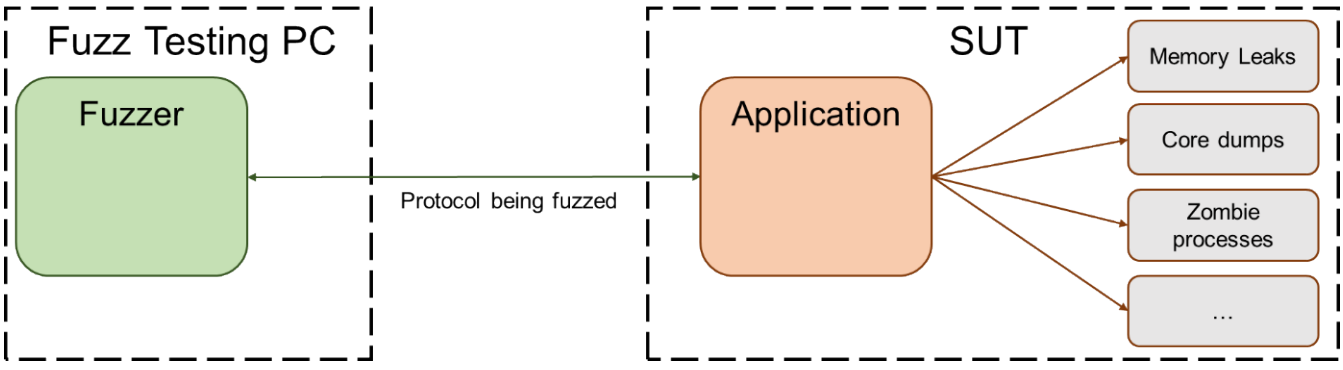


*Figure 1. Examples of issues on the SUT that are undetected over the fuzzed protocol.*

One example of an issue that can go undetected during fuzz testing is memory leaks. During fuzz testing, messages sent over the protocol being fuzzed are processed by the SUT. In this example, the SUT processes the fuzzed messages correctly, and by instrumenting just the protocol being fuzzed, there is no indication of any issues. However, the processing of the fuzzed

messages is causing memory leaks in the application processing the messages. If the fuzz testing is run long enough with perhaps a certain type of message, it might be possible to detect an exception over the protocol that is being fuzzed—for example, a response is slower than expected or a response is missing. However, it would normally be extremely difficult to identify what caused the unintended behavior without having to invest significant time into analyzing all the communicated messages over the protocol. By contrast, being able to detect the memory leak directly on the SUT after the first fuzzed message that causes the memory leak is much more efficient and also allows the developers to pinpoint exactly what caused the memory leak.

Another example of an issue that could be missed during fuzz testing is the case where the application crashes but restarts quickly. In this example, a certain fuzzed message causes the application to crash. But the application restarts quickly, and by the time the instrumentation over the communication protocol occurs, the application seems to be responding correctly and the exception (i.e., the crash) is not correctly identified. From a functional point of view, this behavior may be acceptable since from a user's perspective, everything seems to be working as it should (i.e., the application is up and running fast enough that the crash is not noticeable to the user). However, in the future, this exception may lead to negative effects on the SUT. For example, if it is triggered over time, it might lead to zombie processes or generate core dumps and eventually cause the system to crash or behave in an unintended manner. Alternatively, an attacker might be able to identify the messages causing the crash. If the vulnerability is exploitable, an attacker could craft a specific message that could allow remote code execution. Since the application would not crash, it would also not be restarted, and the attacker would have full control.

When fuzz testing IVIs and telematics units, the above two example issues could lead to vulnerabilities and bugs going undetected. Although fuzz testing is typically considered a black box approach, developers and testers sometimes have access to the internals of the SUT (e.g., Linux and Android), which allows them to run additional scripts on the SUT to help identify exceptions during fuzz testing. That is, it is possible to achieve more efficient and accurate fuzz testing by employing a gray/white box approach where agents placed on the SUT assist with the instrumentation. We present this approach in more detail in the next section.

## Overview of the Agent Instrumentation Framework

The Agent Instrumentation Framework provides the fuzzer with detailed instrumentation data from the SUT during a fuzzing session. This data can be used to determine the fail/pass verdict of a test case. It also provides the tester with valuable information from the SUT.

As no two SUTs are the same, we designed the framework to be modular and allow for the quick creation and adaptation of its functions. The software modules running on the tested system are called Agents. The main purpose of an Agent is to perform a single instrumentation task and feed this collected information back to the fuzzer. However, to automate the fuzzing process and to allow for more complex in-depth instrumentation, an Agent may also execute functions during other times in the execution of a test case.

It is important that an Agent perform no more than a single instrumentation task. If an Agent were to collect too much data on which it bases its verdict for a test case, a tester using the framework would have to dissect the reason for the failure by searching through the log files. Additionally, not every SUT offers the same functionality that can be instrumented. Making the Agent compatible with different configurations and scenarios would slow down testing and possibly require constant modification of the Agent.

We designed the framework with the capability to configure, launch, and control multiple Agents with its own configuration simultaneously. This flexible modular approach gives the user the ability to reuse multiple custom-created Agents, all running with different parameters on the SUT.

## Requirements, architecture, and configuration

The overall main goal of the framework is to find unknown vulnerabilities by using more-advanced instrumentation. Therefore, it is inevitable to shift slightly away from an all-black box approach when testing IVIs and telematics units. Depending on the types of vulnerabilities the user is seeking, the user might need to get access to the operating system running on the SUT so that an Agent can perform its task correctly.

Evidently, from the numerous papers on embedded device security, acquiring access to the operating system of an IVI or telematics unit to allow for the execution of custom code is a trivial task. Remote communication with the OS can be done via the built-in wireless communication technology, serial, or USB to Ethernet converters. Describing how to acquire access to and set up communication with these systems is outside this paper's scope.

The framework has been written in the popular Python programming language, which has a wealth of available libraries that offer great flexibility in writing Agents for the platform. In addition, CPython can be statically cross-compiled to allow the framework to be run on virtually any embedded architecture.

There are two main modes of operation for the framework: synchronous mode and asynchronous mode, each having advantages and disadvantages. They differ in the configuration required in the fuzzing tool, their accuracy in pinpointing a test case to a vulnerability, and their speed of execution. Some Agents can be run in both modes, but the framework can only be configured to use one mode at the time.

The operation mode, the Agents, and their configurations are defined in the main configuration file: `config.json`. The Starter script `start.py` reads this configuration file and starts the processes needed to communicate with the fuzzer and, depending on the mode used, the Agents itself.

## Synchronous mode

Running the framework in synchronous mode gives the tester the greatest control over the Agents running on the SUT, but this mode is also the slowest. It allows the Agent to perform various functions before and after a test case executes, which is sometimes needed for automation and several more-advanced techniques for finding unknown vulnerabilities.

When synchronous mode has been set as the mode of operation in the configuration file, the Starter script launches a daemon on the SUT to wait for incoming requests from the fuzzer. The only requirements on the fuzzer's side are the `client.py` script and a `config.json` file that defines the IP address and port pointing to the daemon process running on the SUT.

Depending on the functionality and requirements of the Agents used, the fuzzer can send information to the daemon at various times during the fuzzing session by calling the `client.py` script with different arguments. These are the available options:

- Before the test run (`python client.py –config config.json before_run`). Because the file contains the information on all Agents to start and their respective configurations, this option is a requirement when running the framework in synchronous mode.
- Before sending a test case to the SUT (`python client.py –config config.json before_case`). This option sets the tested system in a certain mode or start processes.
- After sending a test case to the SUT (`python client.py –config config.json after_case`). This option stops instrumentation process or collection information.
- To initiate the instrumentation process (`python client.py –config config.json instrumentation`). This option requires the Agent to provide the framework with a pass/fail verdict and optional additional verdict information. Depending on the Agent's logic, this function is used to analyze output gathered during the `after_case` phase of the fuzzing session or perform some instrumentation logic itself.
- When a test case has failed (`python client.py –config config.json instrumentation_fail`). This option restores the SUT to a working state. The user could restart a daemon, kill hanging processes, restart a virtual machine, or respawn a Docker container containing the SUT.
- After the test run (`python client.py –config config.json after_run`). Where needed, this option destroys the SUT's environment and shuts down all operations.

The Agents are run in-line with the fuzzing session, which adds overhead to each test case executed, depending on the speed of the Agent's functionality. The user might decide to run an initial set of test cases with fewer Agents configured to get a higher-level indication of SUT failure, then enable more Agents to narrow down the cause of failure to a single issue. Therefore, keeping the Agent's tasks minimal and precise is essential to the overall effectiveness and duration of a fuzzing session.

## Asynchronous mode

Running the Agents in asynchronous mode requires no configuration on the fuzzer's side, other than the ability to parse incoming syslog messages. Rather, the `config.json` present on the SUT defines the asynchronous mode and Agents to be started when the Starter script `start.py` is called. Also defined in the configuration file are the IP address and port of the fuzzer where the syslog daemon is running and, if required, a custom polling rate. By default, the Agents will be polled every 0.5 seconds.

During execution, at the predefined interval, every Agent is asked for the instrumentation data, which contains a pass/fail verdict and additional info to send to the fuzzer. As this instrumentation process occurs periodically, not in-line with the fuzzer sending its test cases, the fuzzer cannot determine the exact test case that caused the failure. Instead, it only reports when a prespecified condition has been met. Therefore, the main advantage of running Agents asynchronously is the speed of execution, with the cost of accuracy.

Consider a buffer overflowing on an infotainment system, slowly building up to a measurable failure over a thousand test cases. Using the framework in asynchronous mode would notify the fuzzer of the failure only at test case 1000, when the actual failure occurs. By contrast, an Agent running synchronously could determine the cause to be, for example, test cases 143, 376, and 1000 (where test cases 143 and 376 indicate some sort of exception but do not yet lead to a full failure). Depending on the type of instrumentation, the user can combine both modes to maximize test run efficiency.

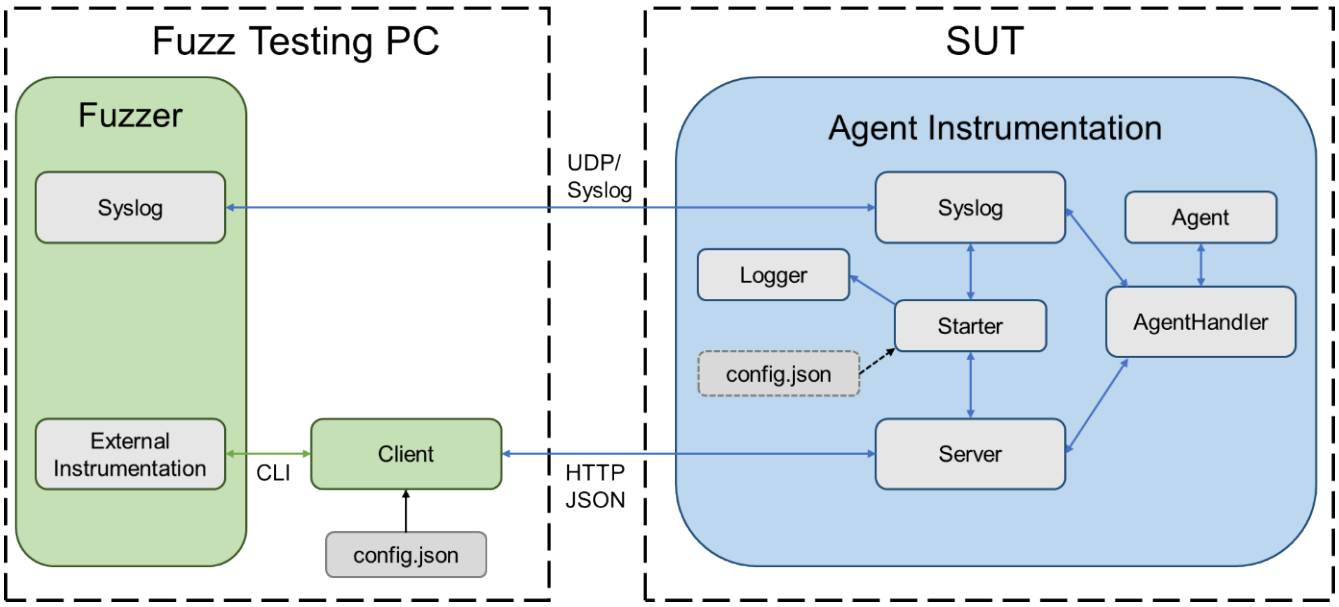


*Figure 2. Various modes of operation visualized.*

# Examples of Agents

While using the framework in multiple fuzzing sessions, we wrote several Agents to perform the tasks we required, extending on the default protocol-specific instrumentation present in fuzzers. The default type of instrumentation usually involved the execution of a valid RFC-complaint sequence of messages in every test case, or even more basic ICMP messages to verify the SUT being alive. The following Agents build on that, providing more insight into what is happening in the SUT.

## AgentCoreDump

The Core Dump Agent looks for the existence of a core dump file. When a process crashes, a core dump can be created, writing the contents of RAM to a persistent storage device. This file can be used to further analyze the state of the process during the crash.

Once the Agent has detected a core dump, it will copy it to a safe location to prevent it from being overwritten. Subsequently, the framework will notify the fuzzer by sending the failed test case notification. Agent parameters that can be set in `config.json` are the path the Agent should monitor and the name of the core dump file.

## AgentLogTailer

The Log Tailer Agent will monitor a log file during its lifetime. If a new line is written to the file and matches any of the predefined parameters, the Agent's instrumentation method will give a fail verdict.

Configurable options are the file to be monitored and the keywords to be matched.

## AgentProcessMonitor

The Process Monitor Agent is used to monitor the state of a process. The Agent can either start the target process or monitor an already-running process. The instrumentation method gives a fail verdict if the target process is down or turned into a zombie process.

Optionally, the Agent can also monitor the process's memory usage and give a fail verdict if the usage goes over a configured limit. The target process is killed and restarted when the `instrumentation_fail` method is called.

Configurable options are the target process's name, memory threshold, and start delay in seconds before the process is respawned.

## AgentPID

Like the Process Monitor Agent, the PID Agent is used to monitor processes running on the SUT, but with more options. Due to the working nature of this Agent, it 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 associated process identifier (PID) and the PIDs of its children. A test case is executed, after which the same mapping is performed again.

If a process has died between the two points of measurement, its PID will not be present in the new mapping. Alternatively, a process might have died but been restarted and issued a new PID, which is common in high-availability configurations where a process daemon watchdog is present. In both cases, a fail verdict is issued, accompanied by information on the event and process itself.

Configurable options are the processes to be monitored.

## AgentAddressSanitizer

The Address Sanitizer Agent can be used to find memory addressability issues and memory leaks in software, using Google's ASAN framework. This allows us to find such issues as these:

- 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

A requirement for the correct operation of this Agent is to compile the target software with additional compiler flags. Similar to the PID Agent, this Agent can only be run in synchronous mode.

This Agent can work in two modes. To find memory leaks exclusively, the user must kill the target process after each test case and analyze ASAN's output. The Agent does this automatically and sets up the environment for the next test case.

The second mode finds all other addressability issues. The Agent will configure ASAN using environment variables to kill the process on finding any issue. As the process can stay running, the speed of this mode is considerably higher than the one for finding memory leaks.

In both cases, the fail verdict is accompanied by a crash trace. If available, detailed crash information will be reported back to the fuzzer.

## AgentValgrind

Like the Address Sanitizer Agent, this Agent finds memory leaks and addressability issues. The difference is the mode of operation and speed. The Valgrind Agent uses various checkers and profilers from the Valgrind project, which effectively emulates a hardware layer for the program to run on. This Agent is quite heavy and adds lots of overhead to each test case; thus, it is not ideal to run on an embedded device, despite its native support for lots of architectures. The ability to instrument processes without the need for recompilation is a large plus over the use of ASAN if time is no limitation or if source code is unavailable.

## An example `config.json` configuration file

Figure 3 shows an example **`config.json`** file.

```
{
    "instrumentation_method": "external",
    "external": {
        "ip": "192.168.1.161",
        "port": 8881,
        "token": "Secret"
    },
    "before_run": {
        "agents": {
                "pid_monitor": {
                "type": "AgentPID",
                "executables": [
                    "/system/bin/mediaserver",
                    "com.android.bluetooth",
                    "android.process.media"
                ]
            }
        }
    }
}
```

*Figure 3: Example `config.json`.*

In this example, the use of the synchronous mode is denoted by the keyword "external," and the **`client.py`** script will send this configuration over to the daemon running on the SUT at 192.168.1.161:8881. A token has been used to use simple authentication to the framework.

The Starter script **`start.py`** on the SUT will load an Agent with the name "pid_monitor" of type AgentPID, configured to monitor three processes named "/system/bin/mediaserver," "com.android.bluetooth," and "android.process.media."

## Implementation and test results

To demonstrate the effectiveness of the Agent Instrumentation Framework in finding unknown vulnerabilities in infotainment systems, we used the framework with various Agents during fuzzing runs. We used the Defensics fuzzer from Synopsys, which allows users to easily extend the instrumentation features offered by default, including protocol-specific RFC-complaint sequences, SNMP, syslog, and functional protocol checks.

Our test targets were infotainment systems from various vendors, including OEM and aftermarket solutions. One of our targets was an extracted operating system running in an emulated environment. The test setup, depicted in Figure 4, was identical in all our fuzzing runs, as we chose to run the framework in synchronous mode for greater control. Some of the Agents we used also required synchronous mode.

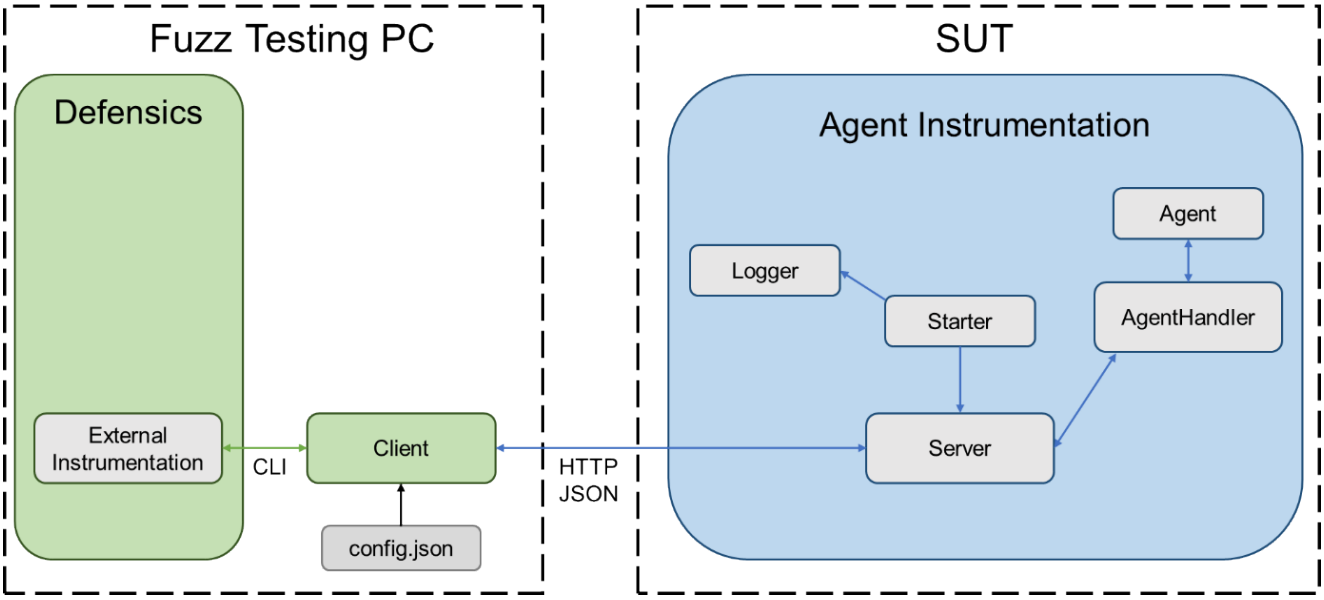


*Figure 4. Setup of the test bench used during the fuzz testing.*

As the synchronous way of running the Agent Instrumentation Framework requires the fuzzer to call the `client.py` script with additional arguments at various times during the session, we need to configure Defensics to do so. Defensics has a built-in feature called external instrumentation that can do exactly that, as shown in Figure 5.

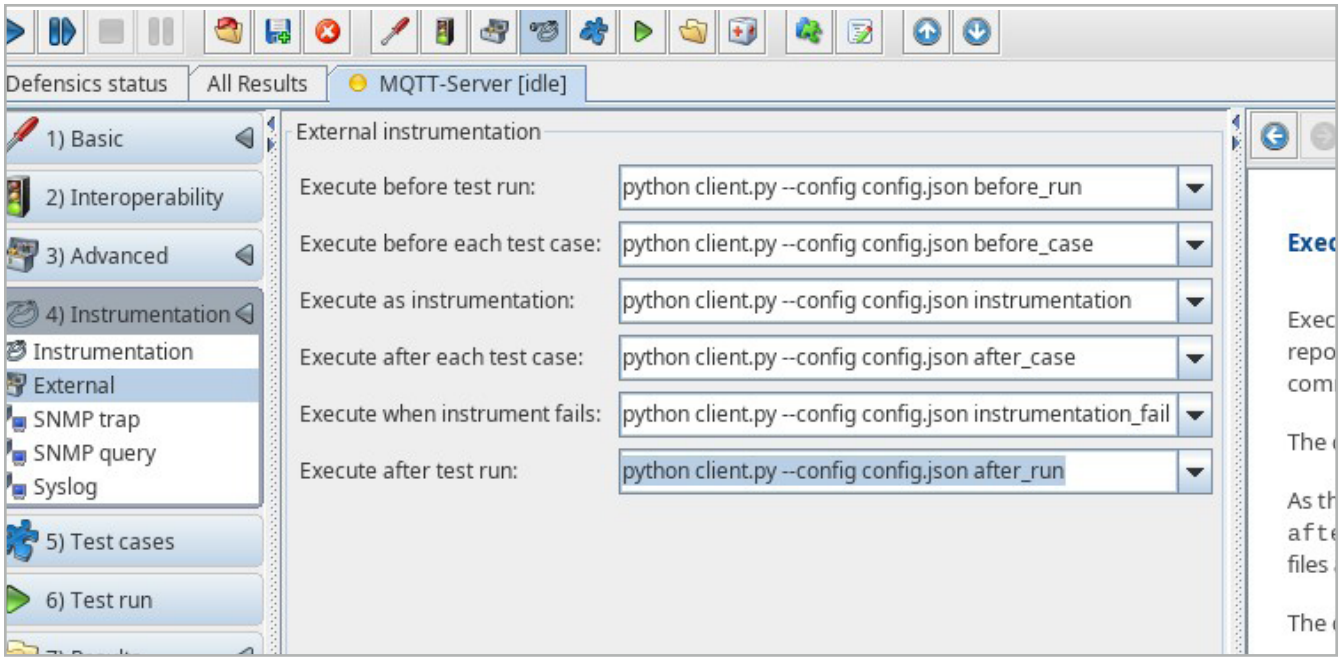


*Figure 5. Defensics instrumentation configured for the Agent Instrumentation Framework.*

In the `config.json` pointed to in Defensics, we configured the Agents we wanted to run on our target with their configuration options. The file also contains the IP address of the framework running on the SUT, so the `client.py` script knows where to send its requests to.

The Agents we used most were AgentPID, which requires no extra configuration or modification of the SUT, and AgentAddressSanitizer and AgentValgrind, where possible, to find memory issues.

A typical infotainment system contains a wide variety of protocols, as the attack surface continues to expand owing to the addition of more functionality and connectivity to the outside world. A full attack surface analysis is outside the scope of this paper, but the interfaces and protocols we focused on were mostly those accessible to the outside world. We highlight a few of these in the next subsection:

- Browser protocols
- Bluetooth protocols
- Wireless (802.11) protocols
- Messaging protocols
- File format parsing (audio, images, video)
- CAN bus

## Bluetooth fuzzing

Defensics includes a scan option for locating and pairing with nearby Bluetooth devices and configuring test suites. For a scan to successfully locate a Bluetooth-enabled device, the device has to be discoverable, an option that has to be enabled manually in most cases.

The scan will return a list of all discovered devices in its vicinity. Further information about the supported services and profiles enabled in the device can be obtained using the service discovery function for the desired device. This information can then be automatically imported into Defensics, and test cases will be generated after an interoperability test of the sequences used by the protocol.

Bluetooth often contains payloads for various applications and runs with low-level rights on an infotainment system. Therefore, its processes and daemons were a good candidate for our Agents to focus on. Bluetooth is also used for critical operations in and around the car, such as unlocking doors and accessing vehicle information.

### Test results

In our test against an infotainment system, we found a critical vulnerability. A single frame containing a buffer overflow anomaly caused the main Bluetooth kernel module to crash. This can be seen by monitoring the bluetoothd daemon process during the execution of the test case, with the use of the AgentPID agent. As the device had a daemon watchdog that quickly restarted bluetoothd, we would not have picked this up without the extra instrumentation in place.

The combination of the anomaly being an overflow case and causing core kernel modules to crash should be cause for concern, as this could potentially lead to further exploitation of the target process running with root rights. The relevant part in the logs showed the PID of the bluetoothd daemon disappearing before it restarted:

```
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]
```

## Wireless fuzzing

The 802.11 protocol family can use the Defensics Monitor WLAN Scan feature, which automatically copies the required parameters to the corresponding settings fields based on the selected target device.

Infotainment systems are often able to spawn a wireless network in the car, allowing passengers to use their 3G/4G connections. Finding new vulnerabilities in the 802.11 implementation is an interesting target, as it could potentially give access to all devices inside the wireless cloud and can be accessed from a larger distance than Bluetooth can.

### Test results

During our testing session, we found a single frame containing a buffer overflow anomaly that caused several kernel modules to crash. Interestingly, this is a nonauthenticated frame and could thus theoretically be sent out by anyone. The combination of the anomaly being an overflow case and causing core kernel modules to crash should be cause for concern, as this could potentially lead to further exploitation of the target system.

Defensics would not have detected this vulnerability, as the kernel watchdog restarted the affected modules immediately without a noticeable drop in connectivity. We could have used the Core Dump Agent if we enabled that functionality in the kernel, or AgentPID to monitor processes affected by the kernel crashes. Instead, we simply used the Log Tailer Agent and found this issue by tailing `syslog` with the keywords “stack,” “crash,” and several kernel module names. The relevant `syslog` output is shown in Figure 6.

```
[  +0.000032] ------------[ cut here ]------------
[  +0.000019] WARNING: CPU: 3 PID: 912 at drivers/net/wireless/[redacted]
[  +0.000002] Modules linked in: loop(O) [redacted]
[redacted]
[  +0.000083] CPU: 3 PID: 912 Comm: [redacted] Tainted: G     U  W  O    [redacted]
[  +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]
...
```

*Figure 6. Syslog output during an 802.11 kernel module crash.*

## Fuzzing MQTT

Lightweight protocols such as MQTT have several advantages over traditional protocols owing to their simplicity and lightweight nature. MQTT is a simple protocol that lets an embedded device publish and receive messages in the cloud. It has minimal packet overhead compared to protocols like HTTP and is therefore very efficient, lending itself to low-power environments. MQTT is also used in automotive components.

As we had access to the source of the MQTT broker we tested against, we could use AgentAddressSanitizer to test for memory addressability issues and memory leaks. To do this, we recompiled the code with additional compilation flags to enable Google's ASAN instrumentation controlled by environment variables the Agent uses during execution.

### Test results

We found a memory leak in the popular MQTT broker Mosquitto using the Address Sanitizer Agent. This vulnerability has since been reported and fixed. When running all test cases generated by Defensics per default, it is impossible to detect this issue without any additional instrumentation.

An example of a correct MQTT Connect message as defined in the RFCs is presented in Figure 7.

mqtt .connect-disconnect .valid - 0x78EF0409DB41550B

Attack Modifier = **0** CVSS/BS = **9.3** (components)

**MQTT CONNECT**

| | | | |
|---|---|---|---|
| 000000 | Fixed-Header | | |
| 000000 | Type | | |
| 000000 | CONNECT | | 4bit 0001 |
| | Flags | | 4bit 0000 |
| 000001 | Remaining-Length | | . 1b |
| 000002 | Variable-Header | | |
| 000002 | Protocol-Name | | |
| 000002 | Length | | .. 00 04 |
| 000004 | Value | | MQTT 4d 51 54 54 |
| 000008 | Protocol-Level | | . 04 |
| 000009 | Connect-Flags | | |
| 000009 | User-Name-Flag | | 1bit 0 |
| | Password-Flag | | 1bit 0 |
| | Will-Retain | | 1bit 0 |
| | Will-QoS | | 2bit 00 |
| | Will-Flag | | 1bit 0 |
| | Clean-Session | | 1bit 1 |
| | Reserved | | 1bit 0 |
| 00000a | Keep-Alive | | .. 00 00 |
| 00000c | Payload | | |
| 00000c | Client-Identifier | | |
| 00000c | Length | | .. 00 0f |
| 00000e | Value | | |
| 00000e | | MQTTServerSuite | 4d 51 54 54 53 65 72 76 65 72 53 75 69 74 65 |
| 00001d | Will-Topic | | () |
| 00001d | Will-Message | | () |
| 00001d | User-Name | | () |
| 00001d | Password | | () |

*Figure 7. A correct MQTT Connect message as found in Defensics.*

The Agent reported several failing test cases, after which it continued execution. The test cases all used a similar anomaly—namely, an underflow anomaly where bytes of the packet were removed before it was sent to the SUT, as illustrated in Figure 8.

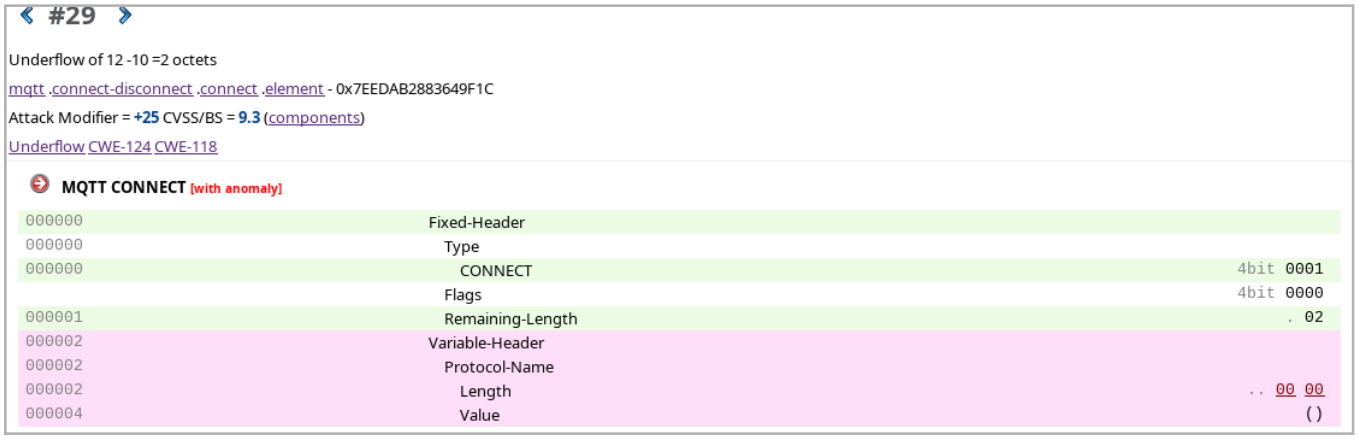


*Figure 8. Example MQTT anomaly in Defensics.*

In each test case, the underflow was a byte larger than the previous. In total, five test cases failed, according to the Agent we used, as shown in Figure 9.

| test-group | index | status | input-octets | output-oct... | diagnosis | time | instrument... |
|---|---|---|---|---|---|---|---|
| mqtt.conn... | 25 | MQTT ... | 4 | 10485764 | pass | 2.060 | 1 |
| mqtt.conn... | 26 | | | 10485763 | pass | 1.179 | 1 |
| mqtt.conn... | 27 | | | 2 | pass | 0.861 | 1 |
| mqtt.conn... | 28 | | | 3 | pass | 0.809 | 1 |
| mqtt.conn... | 29 | | | 4 | fail | 0.995 | 2 |
| mqtt.conn... | 30 | | | 5 | fail | 0.938 | 2 |
| mqtt.conn... | 31 | | | 6 | fail | 0.862 | 2 |
| mqtt.conn... | 32 | | | 7 | fail | 0.867 | 2 |
| mqtt.conn... | 33 | | | 8 | fail | 0.871 | 2 |
| mqtt.conn... | 34 | | | 9 | pass | 0.851 | 1 |
| mqtt.conn... | 35 | | | 10 | pass | 0.170 | 1 |

*Figure 9. Five test cases marked as failing in Defensics.*

The detailed log showed that for each byte increase of underflow, the number of leaked bytes increased by one as well. The Agent also provides Defensics with a trace of the leak, presented in Figure 10.

```
21:34:37 TEST CASE #29
21:34:37 mqtt.connect-disconnect.connect.element: Underflow of 12 -10 =2 octets
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 /home/p0c/synopsys/aif/configs/mqtt-asan.json instrumentation
21:34:37 Instrumentation verdict: FAIL
21:34:37 FAIL Agent: memory_mqtt Info: Agent memory_mqtt says Memory leak found in /home/p0c/mosquitto/src/mosquitto:
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/libsanitizer/asan/asan_malloc_linux.cc:86
21:34:37     #1 0x56218adca9e3 in _mosquitto_malloc (/home/p0c/mosquitto/src/mosquitto+0x3d9e3)
21:34:37     #2 0x56218ade0802 in _mosquitto_read_string (/home/p0c/mosquitto/src/mosquitto+0x53802)
21:34:37     #3 0x56218ade5d85 in mqtt3_handle_connect (/home/p0c/mosquitto/src/mosquitto+0x58d85)
21:34:37     #4 0x56218ade2e77 in mqtt3_packet_handle (/home/p0c/mosquitto/src/mosquitto+0x55e77)
21:34:37     #5 0x56218ade2b61 in _mosquitto_packet_read (/home/p0c/mosquitto/src/mosquitto+0x55b61)
21:34:37     #6 0x56218adca6b7 in loop_handle_reads_writes (/home/p0c/mosquitto/src/mosquitto+0x3d6b7)
21:34:37     #7 0x56218adc891c in mosquitto_main_loop (/home/p0c/mosquitto/src/mosquitto+0x3b91c)
21:34:37     #8 0x56218ada185c in main (/home/p0c/mosquitto/src/mosquitto+0x1485c)
21:34:37     #9 0x7f6af430606a in __libc_start_main (/usr/lib/libc.so.6+0x2306a)
21:34:37
21:34:37 SUMMARY: AddressSanitizer: 1 byte(s) leaked in 1 allocation(s).
21:34:37
```

*Figure 10. Memory leak with added detail provided by the Agent.*

## File format fuzzing

A popular function of infotainment systems is the ability to play rich media content. Simpler systems play only audio file formats, but more expensive systems with larger displays can also display video and image content. The file format parsers on these devices are vulnerable to exploits embedded in the inputs they receive.

Defensics has several file format fuzzers, which can generate fuzzed versions based on the full specifications of various popular file formats. It can simply write these to a disk or use other logic to have these sent to software inputs and determine a verdict.

To test audio and video playback functionality, we created logic to send fuzzed files to various infotainment systems, automatically play these test cases, and assess the verdict using Agents.

# Conclusion

In this paper, we introduced the Agent Instrumentation Framework and explained how it can be used to improve the fuzz testing of IVIs and telematics units. We explained how to better instrument these target systems to allow for more efficient and accurate fuzz testing. One or more Agents deployed on the SUT are used to collect additional information to determine whether test cases on the SUT caused an exception. This information is also provided to the fuzz testing tool and stored in the log file, helping developers identify the underlying root cause of the issues detected and fix problems more efficiently. To show the effectiveness of the proposed framework, we built a test bench based on this approach and performed fuzz testing of several SUTs. We presented our findings and highlighted several examples where issues on the SUTs would not have been detected without agent instrumentation.

The Agent Instrumentation Framework is suitable for IVIs and telematics systems, which are typically based on operating systems providing more functionality, such as Linux and Android, making it possible to run agents on the SUT. We believe that the growth in connected cars and autonomous driving, which continues to drive large volumes of software development in the automotive industry, coupled with an increasing awareness of cyber security in the automotive development process, will lead automated fuzz testing to become a mandatory step for these types of systems. Our proposed framework can help support automated fuzz testing for SUTs running richer operating systems such as Linux and Android.

References


1 D. K. Oka, “Security in the Automotive Software Development Lifecycle,” in *SCIS*, Niigata, Japan, 2018.

2 S. Bayer, T. Enderle, D. K. Oka, and M. Wolf, “Security Crash Test—Practical Security Evaluations of Automotive Onboard IT Components,” in *Automotive—Safety & Security 2015*, Stuttgart, Germany, 2015.

3 D. K. Oka, A. Yvard, S. Bayer, and T. Kreuzinger, “Enabling Cyber Security Testing of Automotive ECUs by Adding Monitoring Capabilities,” in *escar Europe*, Munich, Germany, 2016; D. K. Oka, T. Fujikura, and R. Kurachi, “Shift Left: Fuzzing Earlier in the Automotive,” in *escar Europe*, Brussels, Belgium, 2018.

4 Automotive Grade Linux, Automotive Grade Linux, accessed May 6, 2018; Automotive Grade Linux, Automotive Grade Linux Hits the Road Globally with Toyota; Amazon Alexa Joins AGL to Support Voice Recognition, accessed May 7, 2018; GENIVI, GENIVI, accessed May 6, 2018; Open Automotive Alliance, Introducing the Open Automotive Alliance, accessed May 6, 2018.

5 Synopsys, Heartbleed Bug, accessed May 7, 2020.

6 Oka, Yvard, et al., “Enabling Cyber Security Testing”; Oka, Fujikura, and Kurachi, “Shift Left.”

SYNOPSYS®

# The Synopsys difference

Synopsys helps development teams build secure, high-quality software, minimizing risks while maximizing speed and productivity. Synopsys, a recognized leader in application security, provides static analysis, software composition analysis, and dynamic analysis solutions that enable teams to quickly find and fix vulnerabilities and defects in proprietary code, open source components, and application behavior. With a combination of industry-leading tools, services, and expertise, only Synopsys helps organizations optimize security and quality in DevSecOps and throughout the software development life cycle.

For more information, go to www.synopsys.com/software.

**Synopsys, Inc.**
185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

**Contact us:**
U.S. Sales: 800.873.8193
International Sales: +1 415.321.5237
Email: sig-info@synopsys.com