BLACKDUCK®
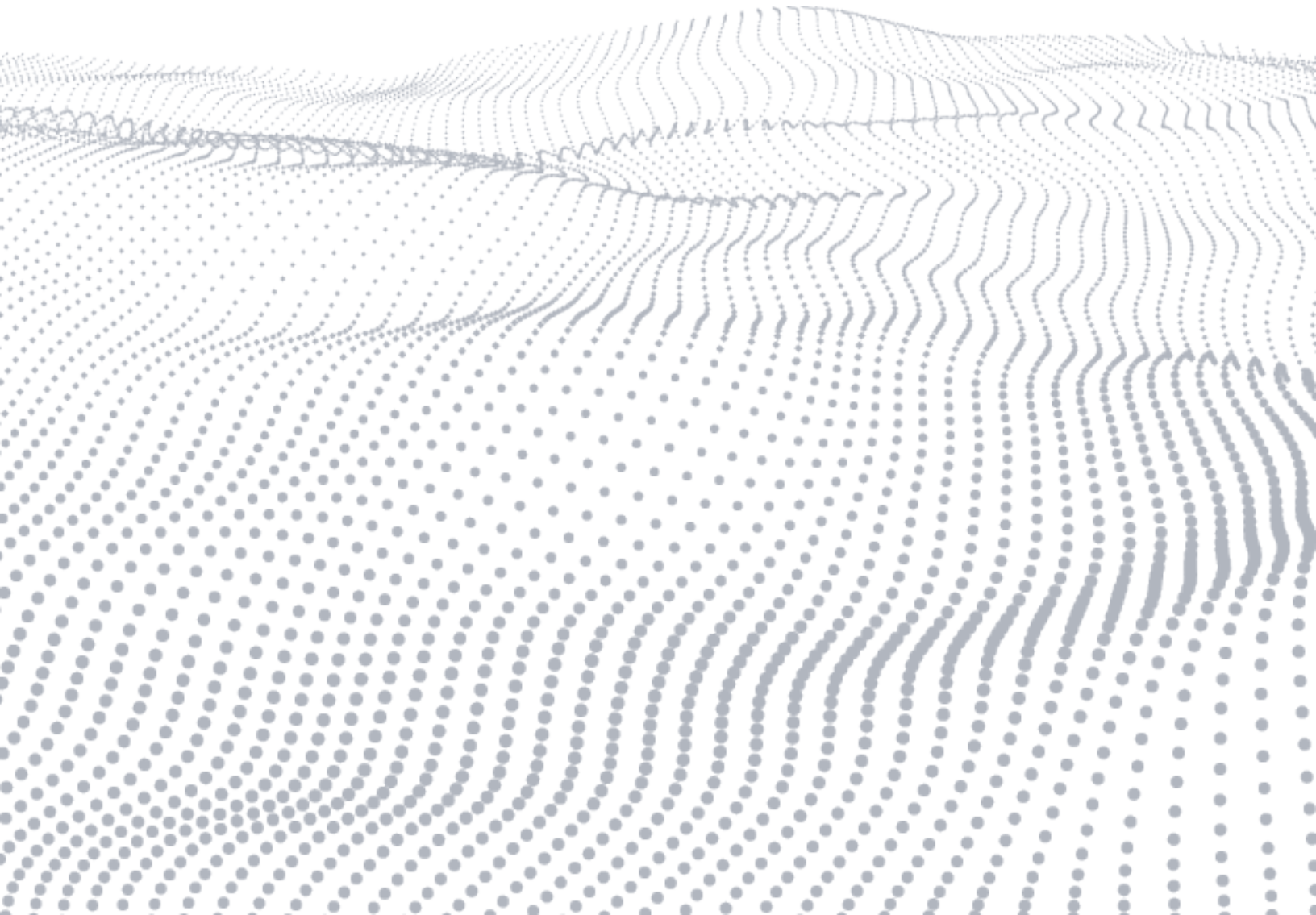
Fuzz Testing Software-Defined Vehicles Using Agent Instrumentation

Overview

In the past few years, cyber security has become 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 software-heavy systems such as high-performance computers (HPCs), several types of issues go undetected, such as memory leaks and cases where the application crashes but restarts quickly.

These automotive systems are based on operating systems such as Linux and Android, so it's possible to collect information from the system under test (SUT) to determine whether any exceptions were detected during fuzz testing. These details about the detected exceptions help developers better understand and identify the root cause of the issues and fix the problems more efficiently.

In this paper, we introduce the Agent Instrumentation Framework and explain how it can be used to improve the fuzz testing of HPCs. We show how information can be collected from the target system to identify exceptions on the SUT to help developers detect the underlying cause of any issues found. We also built a test bench based on this approach and performed fuzz testing against multiple SUTs. Based on our findings, we highlight several examples of issues that would not have been detected without the Agent Instrumentation Framework.

Introduction

There are four major trends driving innovation in the automotive industry: connected, autonomous, shared/services, and electric (CASE). These trends lead to improved safety, comfort, and better user experience, but they also lead to more potentially vulnerable software being used in vehicles, more attack surfaces, and more valuable targets within the vehicles. As such, there is an increasing need for cyber security. The automotive industry has responded with cyber security standards and regulation. For example, ISO/SAE 21434:2021 Road vehicles—Cybersecurity engineering was published August 2021, Automotive SPICE for Cybersecurity was released February 2021, and UN Regulation No. 155 – Cyber security and cyber security management system went into effect January 2021.

In the typical automotive development process, commonly known as the V-model, cyber security is 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 functional testing, vulnerability scanning, fuzz testing, and penetration testing.[1]

Specifically, regarding automotive security testing, "Security Crash Test—Practical Security Evaluations of Automotive Onboard IT Components" provides a comprehensive overview of various testing methodologies.[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 in which 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 increased development in the areas of connected cars and autonomous driving. There is also a shift toward software-defined vehicles, which means vehicle functionalities that had been distributed across a larger number of electronic control units (ECUs) are instead consolidated onto fewer high-performance computers (HPCs). These are software-heavy solutions, so they can contain multiple operating systems such as AUTOSAR Adaptive Platform and Linux/Android, and they can execute a number of different applications, including providing functions for autonomous driving, infotainment, digital cockpit, gateway and connectivity. Since some of these applications will interact with the externals of the vehicle, they are very attractive targets to attackers. They are also more prone to erroneous or malformed input coming from systems or the environment that is out of the control of the vehicle 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 perform fuzz testing on automotive systems; that is, they support the relevant protocols to communicate with the SUT. As such, a simple approach to fuzz testing is to use in-band instrumentation to monitor the communication channel that is being fuzzed. However, one challenge with this approach is that it is often difficult to instrument the SUT 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 determine the underlying root causes for the exception or failure. This information would help 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 hardware-in-the-loop (HIL) systems.[3,4] 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 a controller area network (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 software-defined vehicles, specifically externally facing HPCs such as infotainment systems and connectivity units. We introduce the Agent Instrumentation Framework, which can be used to better instrument these software-heavy 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 based on operating systems such as Linux and Android, [5,6,7] 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.

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 industries such as telecommunications, 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.[8] 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 systems.

But as already mentioned, without proper instrumentation of deeply embedded ECUs using HIL systems, several potential issues go undetected. Similarly, without proper instrumentation of HPCs, numerous potential issues go undetected.

Typically, when fuzz testing a specific protocol, such as fuzz testing over Wi-Fi or Bluetooth on a connectivity unit, instrumentation occurs over the same protocol being fuzzed. In other words, the SUT is being monitored over the same protocol being fuzzed. This limited in-band instrumentation could result in several issues going undetected, as shown in Figure 1.
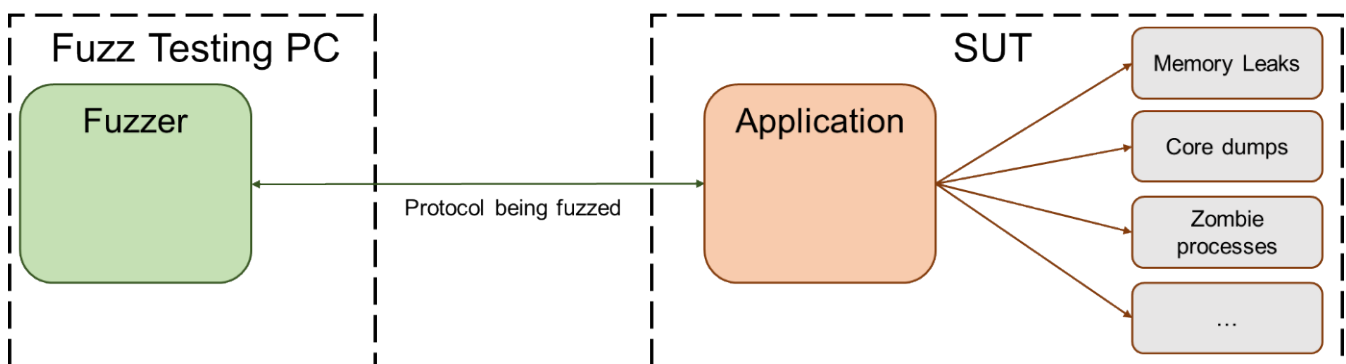

Figure 1. Examples of issues on the SUT that are undetected over the fuzzed protocol.

One 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 this behavior without investing significant time 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 caused the memory leak is much more efficient and also allows the developers to pinpoint the cause.

Another issue that could be missed during fuzz testing is when a certain fuzzed message causes an 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. Functionally, this behavior may be acceptable since from a user's perspective, everything seems to be working as it should—the application is up and running fast enough that the crash is not noticeable to the user. However, this exception may lead to negative effects on the SUT over time, such as zombie processes or core dumps that 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.

These example issues could lead to vulnerabilities and bugs going undetected. Although fuzz testing is typically 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 scripts to help identify exceptions. That is, more efficient and accurate fuzz testing might be achieved by employing a gray/white box approach where agents placed on the SUT assist with monitoring through external 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. 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, increasing the actionability of any found issues.

As no two SUTs are the same, we designed the Agent Instrumentation 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 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 to base its verdict on, 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 and automatically. This flexible modular approach makes it possible to use multiple (custom-created) agents, all running with different parameters on the SUT.

Requirements, architecture, and configuration

The main goal of the Agent Instrumentation Framework is to find unknown vulnerabilities by using more-advanced instrumentation. Therefore, it is inevitable to shift slightly away from a black box approach when testing in-vehicle infotainment (IVI) systems and connectivity units. Depending on the types of vulnerabilities the user is seeking, they might need to access the operating system running on the SUT so that an agent can perform its task correctly.

Numerous papers on embedded device security have made clear that gaining access to the operating system of an IVI or connectivity unit 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 access and set up communication with these systems is outside this paper's scope.

The Agent Instrumentation Framework was written in the popular Golang programming language, which has a wealth of available libraries that offer great flexibility in writing agents. In addition, Golang can be statically (cross-) compiled to allow the framework to be run on virtually any embedded architecture. The framework provides binaries for x86, AMD64, ARM, and ARM64.

The framework's focus is increased accuracy and extended instrumentation features, which enable the fuzzer to pinpoint an event to a single test case. For this reason, the agents included in the framework all operate synchronously to the test case execution.

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, and then enable more agents to narrow 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.

Consider a buffer overflowing on an infotainment system, slowly building up to a measurable failure over 1,000 test cases. Using a fuzzer with protocol-specific instrumentation would only give some indication of failure at test case 1,000, when the actual failure occurs. By contrast, an agent running synchronously could determine the cause to be, for example, test cases 143, 376, and 1,000 (where test cases 143 and 376 indicate some sort of exception but do not yet lead to a full failure).
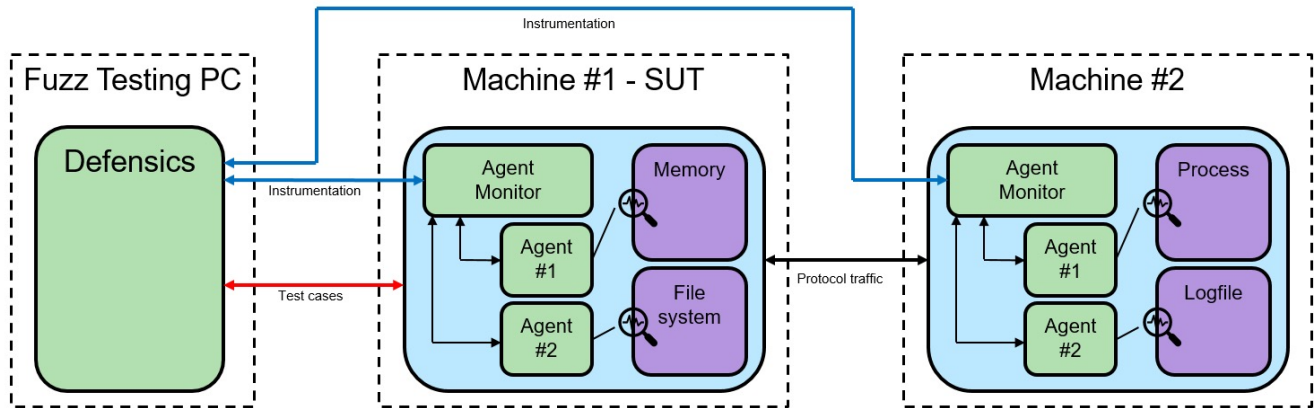

Figure 2. Agent framework architecture

Agent monitor deployment

The Agent Instrumentation Framework is distributed as a single binary for a variety of platform architectures. The binary, when run, will act as a server and listen for incoming connection requests from our fuzzing platform of choice, the Defensics® Fuzzing from Black Duck. Defensics already allows users to easily extend the instrumentation features offered by default, including protocol-specific RFC-complaint sequences, SNMP, syslog, and functional protocol checks. The framework further extends those capabilities.

The monitor server implements a REST API in a web server, which is started by supplying the `server` argument. Traffic between Defensics and the framework is secured via the following options:

• HTTP server is run in a random port.
• HTTP is always protected by TLS.
• HTTP is protected by a self-signed X.509 certificate. The certificate is provided to the client as console argument upon starting the monitor server.
• HTTP is authenticated by a bearer token. The bearer token is provided to the client as a console argument.

The monitor server comes with several configuration options, further detailed in the help test when running the server with the `--help` argument. These options provide greater control over the logging, encryption, and CLI configuration, as well as information on license usage of the project.

```
p0c@h4x0rz ~ $ /opt/Synopsys/Defensics/monitor/agent/agent_linux_amd64
Defensics Agent Instrumentation server provides a RESTful HTTP API for communicating with instrumentation Agents.

See Defensics User Guide for more information on how to connect to the server from Defensics.

Exit codes for different situations:

0: Regular exit without errors
1: Generic error
2: Generic error in server startup
3: Port in use
4: Certificates not available/readable
5: Invalid token provided

Usage:
  agent_linux_amd64 [command]

Available Commands:
  completion    Generate shell completions
  generateCerts Generate certs for HTTP API
  help          Help about any command
  listLicenses  List licenses for third party components
  server        Run Agent server
  version       Prints Agent Framework version

Flags:
      --config string      Config file (default "/home/p0c/.synopsys/agent-instrumentation/agent-instrumentation.yaml")
  -h, --help               help for agent_linux_amd64
  -f, --logformat string   Logging format: text, json. (default "text")
  -l, --loglevel string    Logging level: ERROR, WARN, INFO, DEBUG, TRACE. (default "INFO")
  -w, --workdir string     Work directory for storing data generated during testing (logs etc) (default "/home/p0c/synopsys/agent-instrumentation")

Use "agent_linux_amd64 [command] --help" for more information about a command.
```

Figure 3. Agent monitor configuration options

The agent monitor server can be deployed in the following ways:

- Deployment option 1: On the same system as the fuzzer. This could mean that the fuzzer and the SUT reside on the same machine, or that the agent created by the agent monitor doesn't require presence on the SUT. An agent could interact with an API providing information on which verdict decision are made by the agent's logic.
- Deployment option 2: On the same system as the SUT. This is the most common deployment, as agents created by the agent monitor often require interaction with file systems, daemons, or other OS structures and information sources.
- Deployment option 3: On systems other than the SUT. Often content from test cases impact applications and technologies outside of the SUT. Multiple agent monitors can be started in different locations, all feeding back information to Defensics during a fuzzing session.

Creating agents

Once an agent monitor has been started by calling the binary with the `server` argument, Defensics can connect to it and deploy and configure one or more agents dynamically. These will be automatically started and perform their functions based on instructions from the agent monitor. Depending on the deployment, there are two possible configurations.

- Start local agent server automatically: For deployment option 1.
- Connect to manually started server: For deployment option 2 and 3.

For the second configuration, a CA certificate and an authentication token must be generated by the agent monitor through the `generateCerts` argument. Alternatively, it is possible to use an insecure connection by supplying `--insecure` as argument.

The **Fetch available agents** button will query the agent monitor for all available agent types. It's also possible to create your own custom agents and compile them into the agent monitor binary. Choose the agent from the drop-down list, and it will be created with the configuration provided.
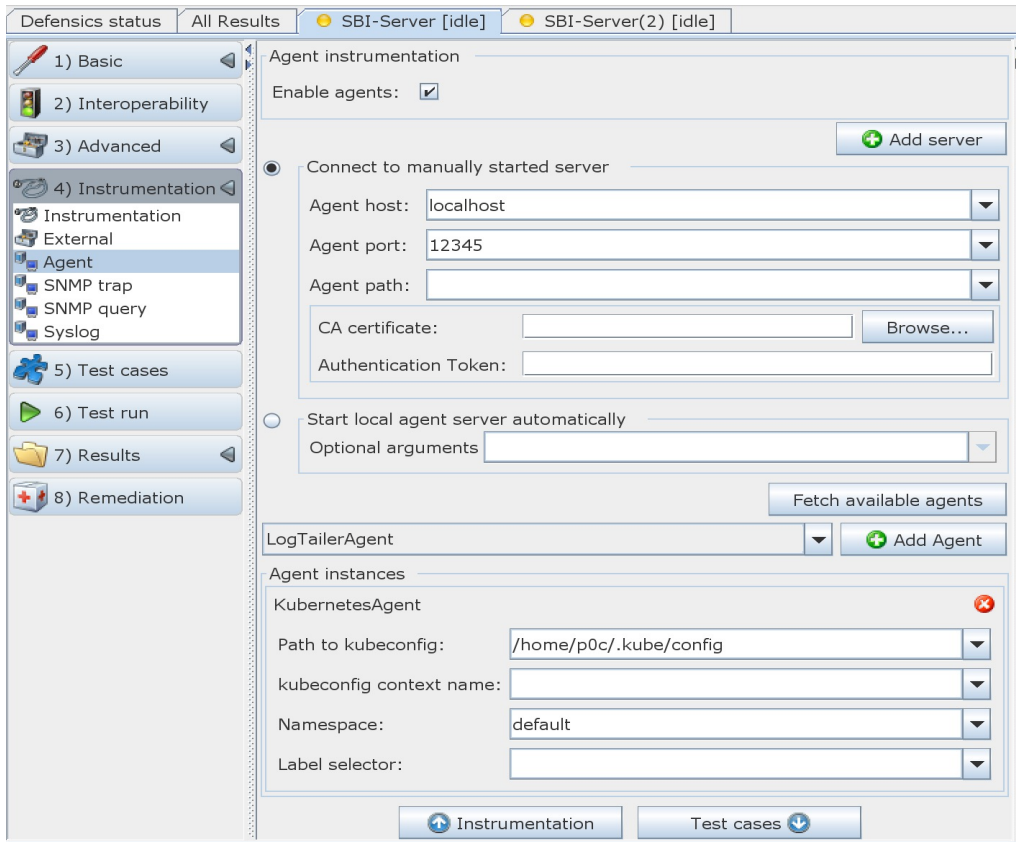
Figure 4. Agent monitor configuration in Defensics

Custom agent development

There are several agents included in the Agent Instrumentation Framework by default, extending the default protocol-specific instrumentation present in fuzzers. The default type of instrumentation usually involves the execution of a valid RFC-complaint sequence of messages in every test case, or even basic ICMP messages to verify the SUT being alive.

You can add your own agents to the framework by implementing them as plugins. The agent monitor communicates with custom agents through a gRPC API using either a UNIX socket (Linux, MacOS) or a TCP socket (Windows) as the communication channel. API calls match instrumentation events (e.g., instrumentation calls an API method for instrumentation). This is implemented using Hashicorp's go-plugin. The framework includes a solution in which the gRPC communication has been implemented already using Golang. You only have to implement the agent interface to add custom functionality to the framework.

Included agents

The Agent Instrumentation Framework comes with several built-in agents. A description of their usage and features is provided here. For more information please refer to the Defensics protocol suite of choice.

LogTailerAgent

Figure 5. LogTailerAgent configuration options

The LogTailerAgent monitors a log file after it has been created. A Golang regular expression can be used to detect errors in the target system. If a line written to the log file matches this expression, instrumentation failure is generated from that line.
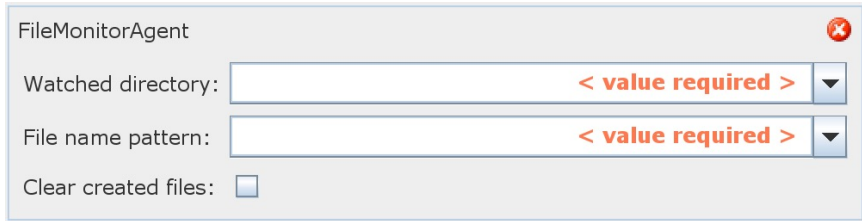
FileMonitorAgent

Figure 6. FileMonitorAgent configuration options

The FileMonitorAgent looks for the creation of files in a directory. A Golang regular expression can be used to match file names. The name of the target file must match the expression for the operation to generate instrumentation failure. For example, if this setting is assigned the value AB, the creation of a file named ABB in the directory specified by the 'watched directory' setting triggers instrumentation failure. The creation of a file named ACC in the same directory does not. If the checkbox is selected, files will be removed and stored in the agent work directory for future reference. The purpose of this setting is to allow testing to continue in cases where an existing file would block files to be created on future errors.
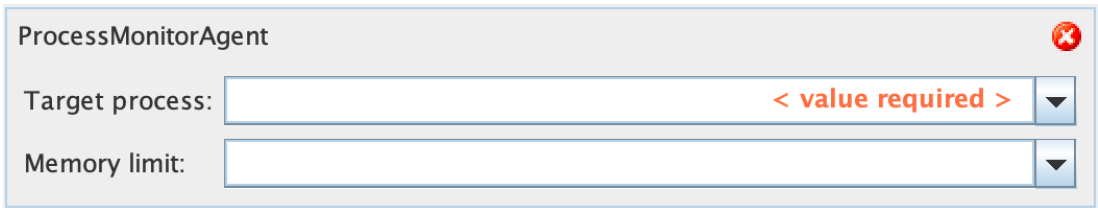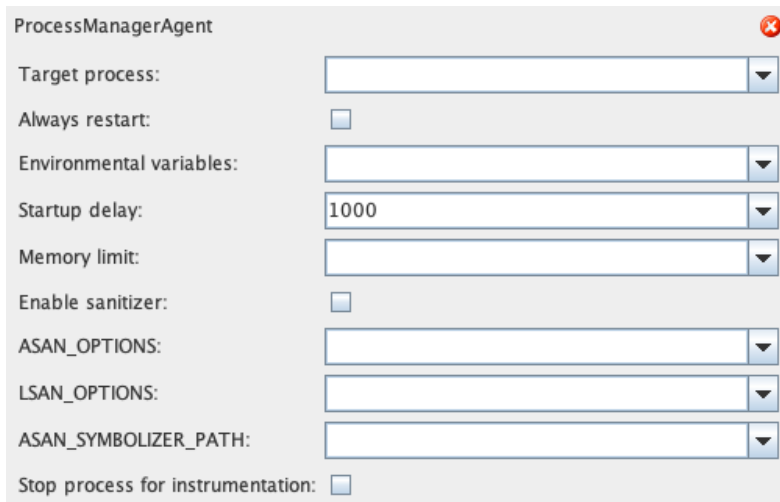
ProcessMonitorAgent

Figure 7. ProcessMonitorAgent configuration options

The ProcessMonitorAgent monitors the state of a process that's already running in the target system. The instrumentation method gives a fail verdict if the target process is down or turned into a zombie process.

The value provided to the Target process setting is matched to the names of the processes running on the target system. Partial matching is supported, so there's no need to specify the full command line of the command.

Additionally, the process can be monitored for exceeding a predefined memory threshold.
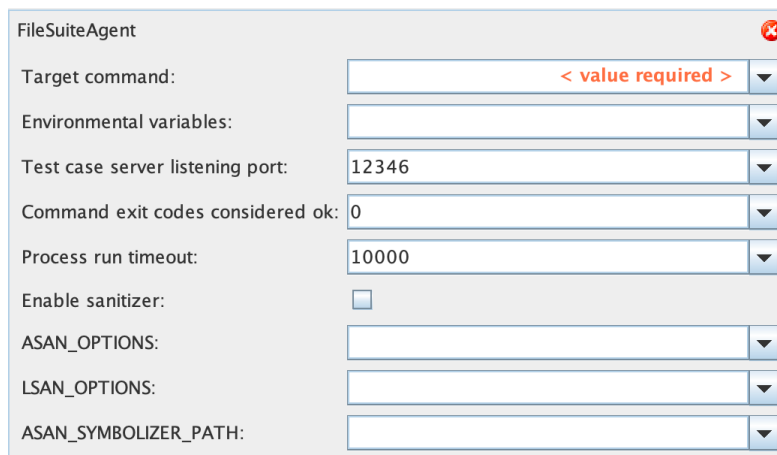
ProcessManagerAgent

Figure 8. ProcessManagerAgent configuration options

ProcessManagerAgent can start a process and monitor it for status and memory usage. Additionally, ProcessManagerAgent can detect memory violations. See the "Sanitizer options" section.

FileSuiteAgent

Figure 9. FileSuiteAgent options

FileSuiteAgent is intended to help with the injection and instrumentation of Defensics file format suites.

There are Defensics suites that generate fuzzed files instead of fuzzed network traffic, and command execution is required to parse these. FileSuiteAgent can automatically receive the Defensics-generated fuzzed content from a TCP connection and write it to the file system, and then execute the target process, passing the received file content as a parameter.

Additionally, FileSuiteAgent can detect memory violations. See the "Sanitizer options" section.

ClientSuiteAgent

Figure 10. ClientSuiteAgent configuration options

ClientSuiteAgent can be used to help testing with Defensics suites, which act as a server for the target acting as client. The agent can be used to execute a command, which triggers target to initiate the connection to the suite at the point where the suite is ready to accept that connection.

Additionally, FileSuiteAgent can detect memory violations. See the "Sanitizer options" section.

Sanitizer options

The sanitizer feature, available in ProcessManagerAgent, FileSuiteAgent and ClientSuiteAgent can find memory addressability issues and memory leaks in software, using Google's ASAN framework. This allows us to find issues such as

• 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

To operate the agents correctly with sanitizer options, you must compile the target software with additional compiler flags, which are only available on Linux and Mac. ASAN and LSAN options can be set for more fine-tuned configuration. In some scenarios it's required to terminate the target process completely, which can be done via the Stop process for instrumentation feature.

This sanitizer feature allows for very precise and in-depth instrumentation of a target application, but it also adds some overhead during test case execution. It is advised to deploy more advanced agents in a parallel test setup and without any other instrumentation when executing large number of test cases.

Figure 11. Sanitizer options in various agents

KubernetesAgent

Figure 12. KubernetesAgent configuration options

The KubernetesAgent allows Defensics to connect to a Kubernetes cluster and monitor pods. This agent does not need to be run on the SUT itself as it communicates with the external Kubernetes API. The agent runs on the same system as the agent monitor and executes its queries from there. The configuration can automatically be read from the kubeconfig file and further filtering can be done through context names, name spaces, and label selectors.

The namespace provided will be monitored for resource changes. Further monitoring limits can be set with the Label selector configuration. Multiple labels can be given as comma-separated list, but if none are given, all supported resources within that namespace are monitored.

The KubernetesAgent can detect pods going down and restarting, which triggers test case failure. As Kubernetes deployments focus often on redundancy, such an event might otherwise go unnoticed. Kubernetes logs are also included in Defensics log files for easier remediation and backtracing.

Implementation and test results

To demonstrate the effectiveness of the Agent Instrumentation Framework at finding unknown vulnerabilities in infotainment systems, we used the framework with various agents during fuzzing runs. We used the Defensics fuzzer from Black Duck, which allows users to easily extend the default instrumentation features, 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.

A typical infotainment system supports a wide variety of protocols, and the addition of more functionality and connectivity continues to expand the attack surface. 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.

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, a setting 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.
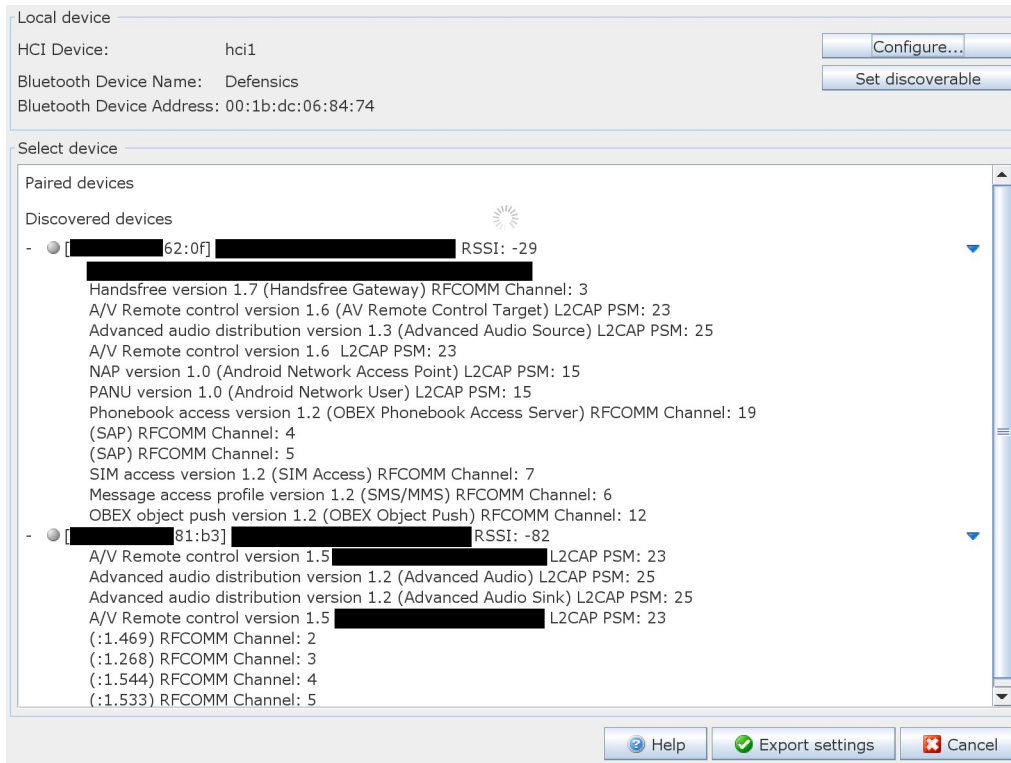
Figure 13. Defensics Bluetooth device scanning dialog

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 ProcessMonitorAgent. The device had a daemon watchdog that quickly restarted bluetoothd, so we would not have picked this up without the extra instrumentation.

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 agent caught the crash and copied stderr output to a local file on the Defensics host for further study.
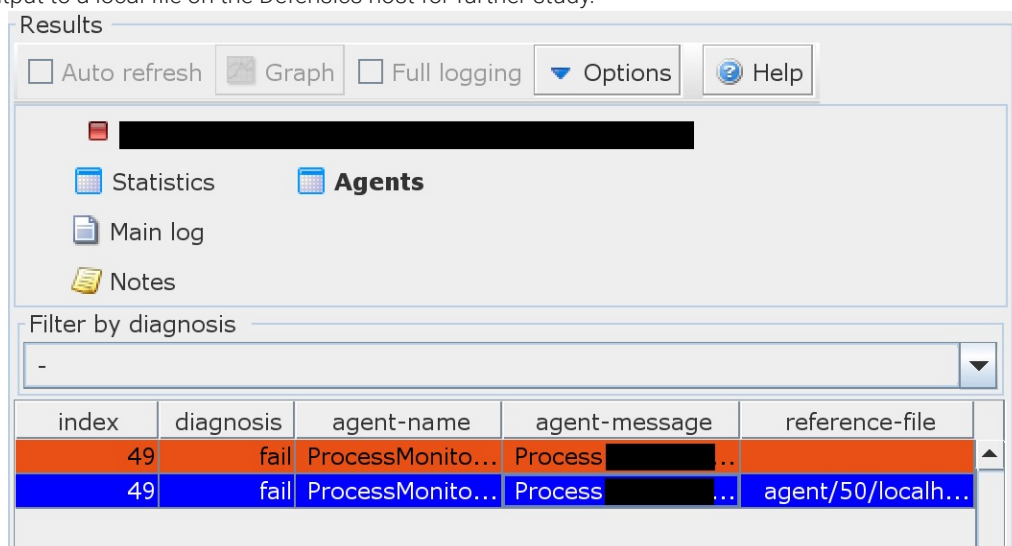

Figure 14. Defensics instrumentation showing agent failure

Wireless fuzzing

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

Infotainment systems can often create a wireless network in the car so that passengers can use their 4G/5G 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 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 so it could 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 caught the creation of a core dump with the FileMonitorAgent if we enabled that functionality in the kernel, or the ProcessMonitorAgent to monitor processes affected by the kernel crashes. Instead, we simply used the LogTailerAgent 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 14.

```
[  +0.000032] ------------[ cut here ]------------
[  +0.000019] WARNING: CPU: 3 PID: 912 at drivers/net/wireless/
[  +0.000002] Modules linked in: loop(O)



[  +0.000083] CPU: 3 PID: 912 Comm:              Tainted: G    U  W  O
[  +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 15. Syslog output during an 802.11 kernel module crash

Fuzzing MQTT

Lightweight protocols such as MQTT have several advantages over traditional protocols thanks 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 the Sanitizer functionality in ProcessManagerAgent 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 Sanitizer functionality in ProcessManagerAgent. This vulnerability has since been reported and fixed. When running all test cases generated by Defensics by 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 15.
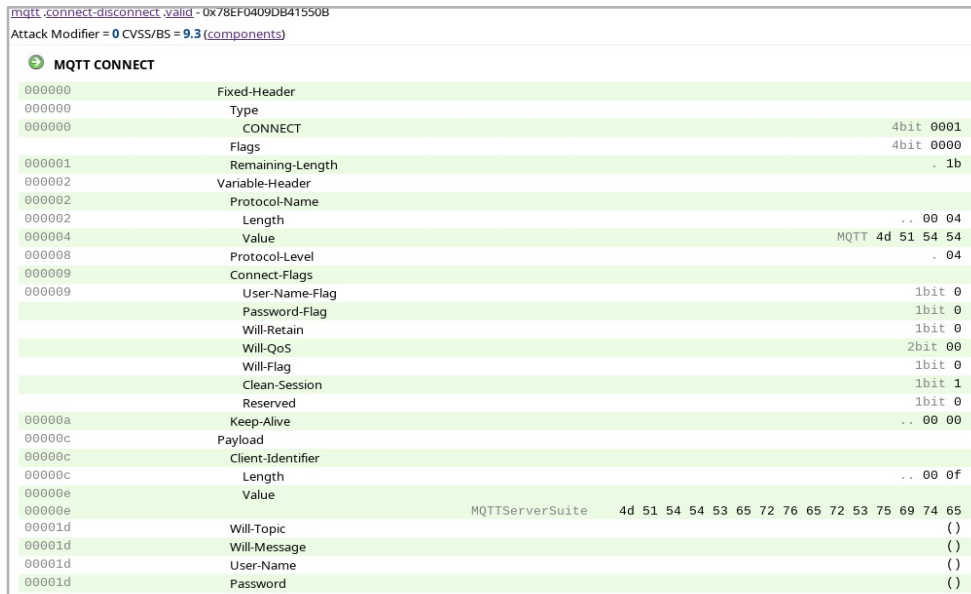
Figure 16. A correct MQTT Connect message in Defensics

The agent reported several failing test cases, after which it continued execution. The test cases all used a similar anomaly—an underflow anomaly in which bytes of the packet were removed before it was sent to the SUT, as illustrated in Figure 16.
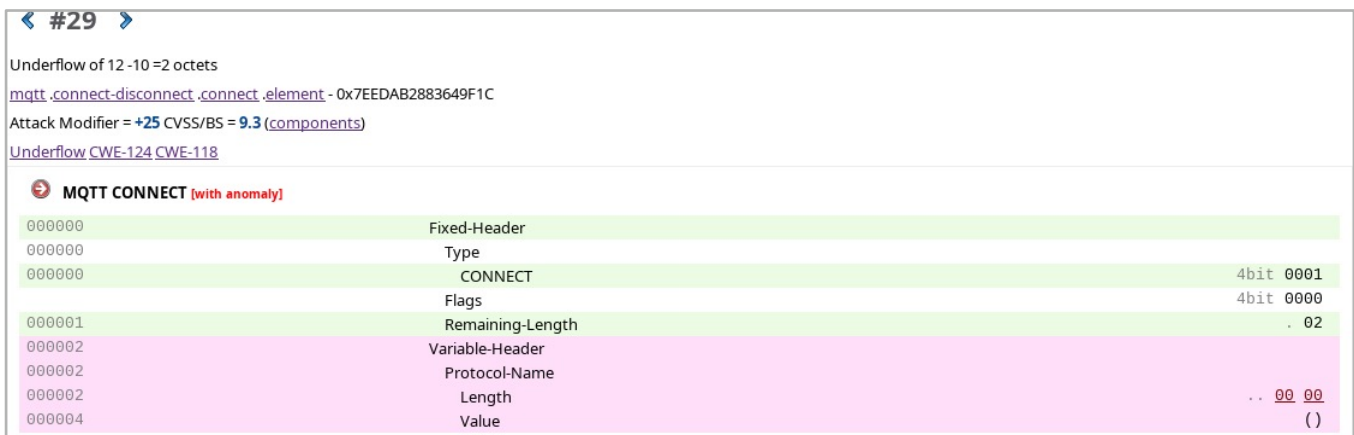

Figure 17. Example MQTT anomaly in Defensics

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 17.

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 18. 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 18.

```
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 19. 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 play 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 that 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.

Test results

To test audio and video playback functionality, we used Defensics built-in file format logic to send fuzzed files to various infotainment systems and automatically play them, and then determined the verdict using agents. A combination of agents could have been used here to instrument various sources, but to keep our setup as minimal as possible we opted for the FileSuiteAgent.
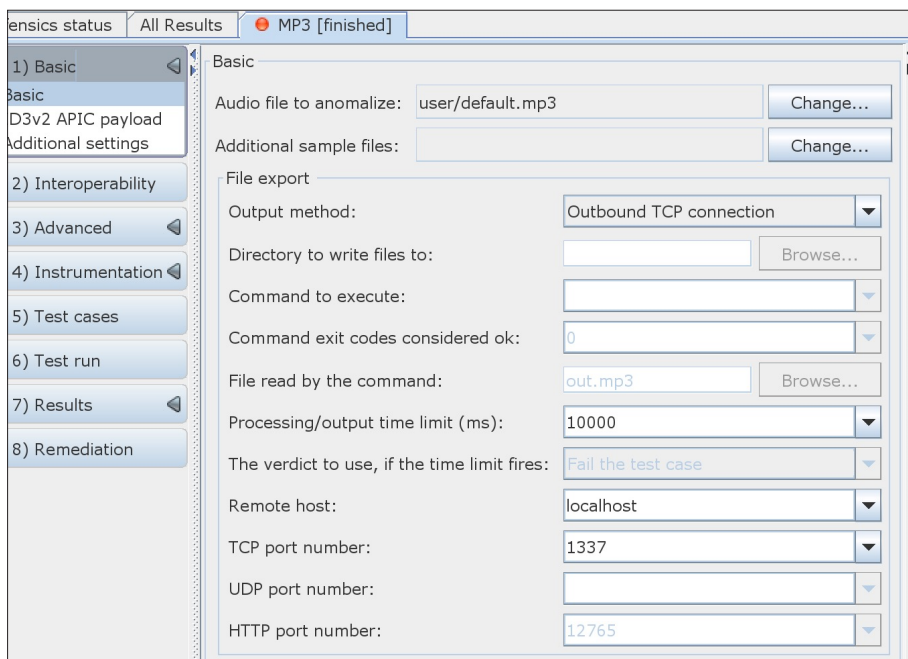

Figure 20. Defensics MP3 file format suite configuration

The configuration of the file format suites is very straightforward. Each generated (fuzzed) file will be exported with the option Outbound TCP connection, which means that it will be sent over TCP to the specified IP address and port combination. The port here is the same as the one configured in our FileSuiteAgent running on the SUT.
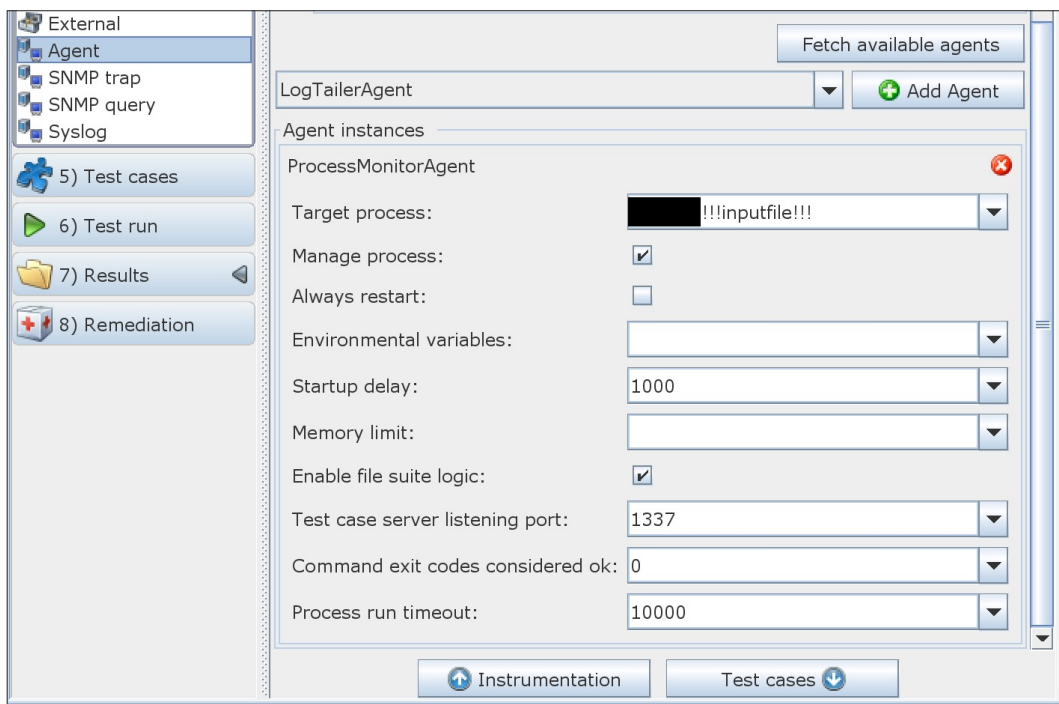

Figure 21. ProcessMonitorAgent configuration

When a test case is executed, the file will be transmitted to the agent running on the SUT. We also specified the command required to have our target process parse the fuzzed file. In the Target command option, we used "!!!inputfile!!!" as a placeholder for the file name used as an argument to the target process, which is handled by the agent's logic during execution. The instrumentation is automatically done by the agent's default configuration.
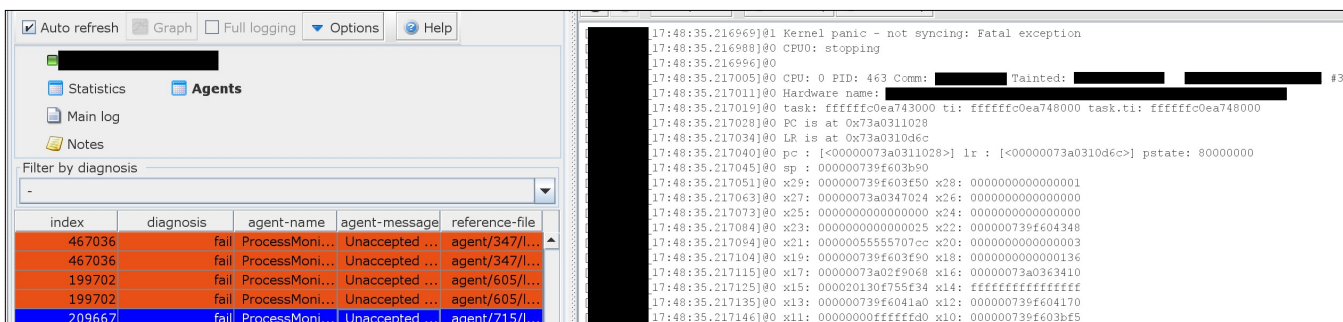

Figure 22. ProcessMonitorAgent's reported failing test cases

During our testing effort we observed several crashes. The effects on the infotainment systems we tested included process crashes, kernel panics, and in one case, a full reboot. These severe issues were found while running only a small subset of the test cases available in our audio, video, and image file format suites.

Conclusion

In this paper, we introduced the Agent Instrumentation Framework and explained how it can be used to improve the fuzz testing of HPCs, specifically focusing on infotainment and connectivity 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 can be 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 HPCs, which typically use 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, "Building Secure Cars: Assuring the Automotive Software Development," Wiley, 2021.

[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, 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*, 2016.

[4] D. K. Oka, T. Fujikura, and R. Kurachi, "Shift Left: Fuzzing Earlier in the Automotive," in *escar Europe*, 2018.

[5] Automotive Grade Linux, Automotive Grade Linux, accessed May 6, 2018.

[6] Automotive Grade Linux, Automotive Grade Linux Hits the Road Globally with Toyota; Amazon Alexa Joins AGL to Support Voice Recognition, accessed May 7, 2018.

[7] GENIVI, accessed May 6, 2018; Open Automotive Alliance, Introducing the Open Automotive Alliance, accessed May 6, 2018.

[8] Black Duck, Heartbleed Bug, accessed May 7, 2020.

About Black Duck

Black Duck® offers the most comprehensive, powerful, and trusted portfolio of application security solutions in the industry. We have an unmatched track record of helping organizations around the world secure their software quickly, integrate security efficiently in their development environments, and safely innovate with new technologies. As the recognized leaders, experts, and innovators in software security, Black Duck has everything you need to build trust in your software. Learn more at www.blackduck.com.