Today’s motor vehicles are increasingly being integrated into the Internet of Things (IoT). Vehicles are connecting to devices, other vehicles, infrastructure and cloud services via new interfaces. On the one hand, this enables new functions and business models. However, this added connectivity also opens more potential areas of attack related to cyber security threats. The rapid introduction of new and complex functions harbors the risk that development teams will hardly be able to keep up with the new attack vectors that result and safeguard against them. General guidelines for developing secure vehicle systems are described in the Cybersecurity Guidebook for Cyber-Physical Vehicle Systems (J3061). It contains strategies and techniques for preventing and detecting security breaches, and it lists measures to mitigate the undesirable effects of a successful attack. Although development processes for vehicle security have become better and are using increasingly more detailed threat models, there is still a gap in the development tools regarding the identification and prevention of security vulnerabilities in existing software.

In ECU tests (Figure 1), the focus is on validating functional requirements and error states within a defined scope – which is limited by the boundaries of the selected parameters. As soon as all functional requirements have been tested sufficiently, it is also necessary to work in the security area with system inputs which are not part of the “expected” parameters. Vulnerabilities lurk in this complementary set of inputs, and they are just waiting to be exploited. Due to time constraints, it is not possible to test

---

Fending Off Cyber Attacks – Hardening ECUs by Fuzz Testing

In designing vehicle communication networks, security test procedures play an important role in the development process. Fuzz testing, which originated in the traditional IT field, offers automatable tests which are used to efficiently check an ECU’s security-related capabilities. But how can test cases be generated using familiar exchange formats?
Compared with other security test methods such as penetration tests, this approach offers a good cost-benefit ratio. Fuzzing engines can be categorized according to three criteria:

- Generative engines and mutation engines: Generative engines generate test data programmatically, while mutation engines generate test data by mutating known valid data.
- "Simple" and "intelligent" fuzzing engines: In contrast to "simple" engines, intelligent engines are familiar with the structure of the data. However, one engine is not necessarily better than the other. An entirely "simple" engine, for example, can send fully random data to a system without requiring any knowledge of the system states or system parameters, like a data checksum in a message. An entirely "intelligent" engine would follow the protocol and system specifications 100%, so it could potentially never detect errors. A good middle path is to use a system that intelligently chooses which system parameters and logical approaches to subject to fuzz testing.
- The third classification criterion is knowledge about the system states and program structure. Here, the fuzz tests check either a black box, white box or gray box system. For some system properties, a certain level of knowledge of the system state is essential. This applies all possible input values, especially when they depend on different states and conditions. Therefore, the goal of fuzz testing is not to test all the input parameters of a system completely, but instead to programmatically generate security tests that are more likely to identify implementation errors.

What is Fuzz Testing and What Benefits Does It Offer?
The purpose of fuzz testing is to detect unknown vulnerabilities by stressing a system with queries which are in part random and quite inappropriate. This concept is not only applied in security tests, but in robustness tests in general. Fuzz tests go far beyond the typical tests developers perform to study all reasonably predictable manipulations. They involve generating additional test cases which ideally come up with fully unexpected data. Randomly generated input data that is applied over many test runs attains a level of test coverage that is hardly attainable by manually created tests.

Since fuzz testing is the technique preferred by hackers for revealing vulnerabilities, developers need an easy-to-use extension for test tools which they can use to meet the challenges of attackers head-on. Ideally, fuzz testing will support a method of test case generation that is largely automated and can indicate problems “relatively” quickly.

Figure 1: Fuzz Testing as part of the V-model for developing security-related ECUs
A successful fuzz test solution must keep records of the test cases that are executed along with all relevant parameters. This lets testing engineers replicate the tests to compare results based on clearly defined test cases and parameters.

The test tool should deliver reports that lend themselves to implementing corrective actions as well as relevant summaries that present additional insights to test engineers.

Currently, there are some promising smaller vehicle-specific prototypes and IT-specific fuzzing solutions available. However, there is no standard automotive fuzzer for vehicle tests which fulfills all the mentioned requirements. The signals in vehicle networks pose a real challenge in the development of fuzz tests. In generating inputs that can be processed by ECUs, fuzzing tools which are popular in the IT sector cannot always fulfill requirements such as floating-point numbers, signed and unsigned N-bit integers, signal multiplexing and changing byte orders in the signal.

Requirements for Successful Fuzz Testing Solutions

Based on own experiences and discussions with developers of vehicle tests and security service providers, Vector has defined key requirements for successful fuzz testing solutions in the automotive industry.

- Automotive OEMs and suppliers generally have full access to databases in which the communication of the entire vehicle network or of a subsystem is described. Defined in these databases are regular messages, expected value ranges, lengths, timing information and data types. These descriptions exist in standard data exchange formats such as DBC, LDF, FIBEX and AUTOSAR System Extract. The descriptions are an excellent source of information for configuring fuzz test generators, which automatically generate test cases.

- A successful fuzz test solution must keep records of the test cases that are executed along with all relevant parameters. This lets testing engineers replicate the tests to compare results based on clearly defined test cases and parameters.

- The test tool should deliver reports that lend themselves to implementing corrective actions as well as relevant summaries that present additional insights to test engineers.

Concept for Automatically Generating and Executing Fuzz Tests

How can the described requirements be fulfilled? The core idea is to use vehicle communication databases to automatically generate and execute the fuzz tests (Figure 2). The overall concept consists of a system under test (SUT), a communication database, a vehicle test framework and an engine for creating fuzz tests. The SUT might be a remaining bus simulation, an individual ECU, a complete subsystem of multiple ECUs or a complete vehicle system.

![Figure 2: Concept for automatically generated fuzz tests](image-url)
A combination of simulated and real elements is also possible. The SUT is connected to the vehicle test framework via one or more communication channels. Vector uses the communication database as the central supplier of input data for the vehicle test framework. Among other things, the test framework can visualize the message contents, definitions of the network nodes and parameterization of the communication channels allocated to them. The user can now select the signals to be subjected to the fuzz test and store them in a configuration. Subsequently, the test framework of the fuzz test engine supplies the interfaces (APIs) for bus access based on its knowledge of the protocol, data types and conversion rules that are used. The fuzz test engine generates the test values for the SUT based on the configuration created using the vehicle test framework. The strategies for generating the test vectors depend on the specific capabilities of the fuzz test engine. The generated test data is then passed to the vehicle test framework, which packs the data for the relevant bus or communications channel and sends it to the SUT.

Implementation Using Standard Software
The described fuzz test framework was created by using standard software and extending it (Figure 3). Serving as the test framework is the testing and development software CANoe, which is used by many ECU testers and offers user-friendly test flow control. Moreover, it fulfills requirements for processing vehicle communication protocols and communication database formats with its libraries, APIs and interfaces. Based on the imported database, CANoe visualizes the network and suggests a list of signals the user can select from. It then generates the configuration of the fuzz test engine according to the user’s selections. This is based on the vTESTstudio test design tool. The engine generates the signals – based on the configuration and its own fuzzing algorithms – which CANoe then prepares as a message that it sends to the SUT over the vehicle bus. Fuzz test execution is logged in such a way that all the test messages can be examined. Because the fuzz tests are integrated in CANoe, that tool’s internal logging function can be used to record messages received by the SUT and those sent to it. All test cases exhibiting unexpected behavior are highlighted in the test window (Figure 4). The created test cases are formulated in CANoe’s own CAPL programming language, and this makes it possible to view them and edit them if necessary.

Challenges
It must always be remembered that fuzz testing is not a comprehensive security solution; rather it is a brick that is used as part of the solution. Although fuzzing engines are continually becoming more refined, and there are new monitoring approaches, this technology as a whole still faces some challenges. In fuzz tests, errors can go unnoticed if they do not cause a complete program crash. Code should be developed so that assertions are implemented...
correctly in all software components, and they support investigation into the causes in case of a failure. The analysis of test cases leading to a crash is may be difficult. Depending on the available monitoring, and in software with complex inputs it takes much more effort to create fuzz tests that provide sufficient code coverage.

Advantages of Fuzz Testing
Fuzz testing offers far-reaching automation that can reveal critical system defects which would remain hidden in manual security audits and software audits. Fuzz testing also provides an overall picture of software robustness. This helps to identify security vulnerabilities and to correct them before they can be exploited for attacks in the field. When fuzz testing is used early in an ECU’s development cycle, security gaps can be revealed early on when it is less costly or time-consuming to correct them. In combination with existing tools and test frameworks from the automotive industry, fuzz testing can make an effective contribution toward validating vehicle software. The option of programmatically generating fuzz tests from existing databases makes it possible to test signals as well as state-dependent protocols.

Translation of a German publication in Hanser automotive, special edition “Safety & Security” August 2018

Image rights: Vector Informatik GmbH