Testing Security of Embedded Software Through Virtual Processor Instrumentation

Andreas Lauber(✉) and Eric Sax

Karlsruhe Institute of Technology, Engesserstr. 5, 76131 Karlsruhe, Germany
{Lauber,Sax}@kit.edu

Abstract. More and more functionality that demands remote access on a vehicle is integrated into modern cars. Fleet management, infotainment, updates-over-the-air and the upcoming functionality for autonomous driving need gateways that enable a car-2-x communication. Misuse is a threat. Consequently, security mechanisms play an increasing important role. But how can we show and prove the effectiveness of these security functions?

Therefore, in this paper we will show an approach to test security aspects, based on virtual instrumentation. The approach is to use a framework that executes the application under development on a virtual model of the target micro controller. An interception library generates scenarios systematically, whereas the effects on registers and memory are monitored. We are intercepting the running software at vulnerable functions and variables to detect potential malfunctions. This will detect security vulnerabilities of all internal failure even if no malicious behavior at the interfaces occur.

Keywords: Virtual processor · Security · Testing

1 Motivation

Within the last decade mobility has undergone major changes. One is the advent of data exchange between cars and infrastructure. Instead of a vehicle being a standalone mechanical device it has been transformed to a mobile platform with extensive electronic sensors and computing power. Nowadays within a car a large amount of data is available in form of sensor data, representing the state of the vehicle as well as its understanding of the surrounding. By exchanging such information with others, new concepts for efficient driving, optimizing traffic flow (see Kramer in [1]), and new comfort functions become possible.

On the other hand many new threats are generated. The increasingly technology allows the transmission of large amounts of data in real time to transmit states for diagnosis and software updates over the air will be possible. I.e. the EE topology will get accessible via an air interface. Therefore the vehicles may offer new attacking surfaces, as some examples already show today.

It has already been shown how modern cars can be attacked and controlled without having physical access to these vehicle [2, 3]. These attacks allow the manipulation of car's brakes and driver assistance systems or remote eavesdropping on conversations

© Springer International Publishing AG 2018
M.E. Auer and D.G. Zutin (eds.), *Online Engineering & Internet of Things*,
Lecture Notes in Networks and Systems 22, DOI 10.1007/978-3-319-64352-6_9

held within the car. They are just a few examples of possible attacks. With an even more connected car an even broader attack vector might be created.

To secure the vehicles against possible attacks security mechanism needs to be implemented which has been a research focus within the last couple of years. Unfortunately not all attack surfaces can be closed by integrating security mechanism. Attack vectors can also arise through sloppy implementations and inexperienced programming. To overcome these issues functional testing and testing for security weaknesses are necessary.

This paper is structured as follows: First we give a short overview of state of the art security testing in Sect. 2. Afterwards we categorize the attacks on systems and point out the important test cases in Sect. 3. Thereafter the virtual instrumentation and processor interception is explained in Sect. 4. The interception lead to the security testing framework shown in Sect. 5. At the end we conclude with a summary and give an outlook for future work.

2 State of the Art for Security Testing

2.1 Theoretical Security Analysis

In theoretical security analyzes one must distinguish between high-level design analyzes and detailed analyzes. In design analysis, protocols, interfaces, and specifications are analyzed by reviewers to find and resolve systematic vulnerabilities such as bad encryption or short keys. While only a theoretical description of the system has to be present during design analyzes, one needs explicit knowledge about the implementation of the algorithms for the detailed analyzes.

Theoretical security analyzes cannot detect any implementation errors due to misinterpretation of the specification or errors from third-party software. To protect the system against this type of error, Bayer recommends in [4] secure software development standards. These can be achieved by means of the various standards and coding guidelines. Even if errors are reduced, errors by specification or errors in third-party software, can only be found by explicit tests of the functions in the overall system.

2.2 Static Code Analysis

For static code analysis the source code is automatically analyzed by means of formal criteria, to identify volatility errors. Static code analysis can identify implementation errors, but functional errors or design errors cannot be found by this analysis. In addition, Knechtel described in [5] that these kind of analyzes are unreliable. He suggests the use of explicit code reviews for sensitive functions. Another option to find weaknesses is the system test on the real platform or a hardware prototype.

2.3 Functional Security Testing

Functional testing serves to ensure the correct execution of algorithms. Spillner describes in [6] how software can be tested. A careful execution of the tests can detect implementation errors and the resulting security vulnerabilities at an early stage. In order to ensure fault-freeness of the tests, official security tests cases are usually carried out, which also cover typical limits of the algorithms.

The algorithms are tested not only for the correct behavior according to the specification, but also for the robustness of these tests. In addition, the performance of security algorithms is tested to identify potential bottlenecks that could affect overall security performance, according to Bayer [4].

However the functional security testing will test the security algorithm as standalone functions. An interaction with other functions of the system is not performed. Therefore a weakness of the outer or sub function will affect the security of the system even if the security algorithms are well tested. This means the security test should always include functions of the overall system.

2.4 Fuzzing for Security Testing

Fuzzing is a technique that has been used for some time to test software in IP networks. To do this, the implementations are subjected to an unexpected, invalid, or random input, with the hope that the target will react unexpectedly to identify new vulnerabilities. The responses to such attacks range from strange behavior of interfaces, unspecified behavior of the system to complete crashes.

As a rule, the fuzzing can be divided into three steps. First, the input data is generated. This can either be structured according to the specification or completely random. The data are then fed into the system interfaces and the output is monitored. As a last step, the recorded behavior must be analyzed by experienced programmers in order to identify potential weaknesses. Disadvantage of fuzzing is that only the interfaces of the system are monitored. Faulty states within the system cannot be detected.

2.5 Penetration Tests

While the above tests can be automated, the penetration test is a test method using human testers. These tests attempt to exploit known vulnerabilities and gain access to the system. The appropriate approach is usually based on years of experience by human testers who perform these tests. An example of typical penetration tests is exploiting undocumented debugging interfaces to gain access to buses and internal signals, but also by opening the controller and directly accessing the silicon, the testers are looking for information on possible attack vectors, according to Bayer in [4]. The knowledge provided for the testers usually range from no information, access to the specification to all information about the source code. Therefore, these tests can be used for black box, gray box and white box tests.

The state of the art security testing can usually only be automated for independent functions without the interaction of all functions in the complete system. Knechtel writes

in [5] that attacks are rarely due to weaknesses of individual keys or algorithms but rather by weaknesses of the entire system. I.e. for security testing of the overall system, including third party software, the overall system needs to be present. Further the internal state of this system needs to be monitored in addition to the external interfaces. This leads us to use virtual instrumentation of a processor running the software under test.

Finally, it should be noted that all practical security tests cannot guarantee complete coverage. Therefore a compromise between test effort, time and completeness must be made. I.e. practical security test serve only as a complement to theoretical security analyzes and the consideration of security in the design phase.

3 Categorization of Attacks

As Radzyskewycz writes in [7], it is not a question of whether systems are attacked, but when. Therefore it is important to implement security mechanisms according to the state of the art. In addition, Wheatley in [8] describes that 44% of all attacks will be done over known vulnerabilities.

The Symantec cooperation describes in [9] the loss or theft of passwords, incidental ties and insider knowledge as other important causes of intrusion into systems. Only a very small part of the attacks on systems are carried out by unknown vulnerabilities at the time of attack. A distribution of the given causes for attacks is shown in Fig. 1.

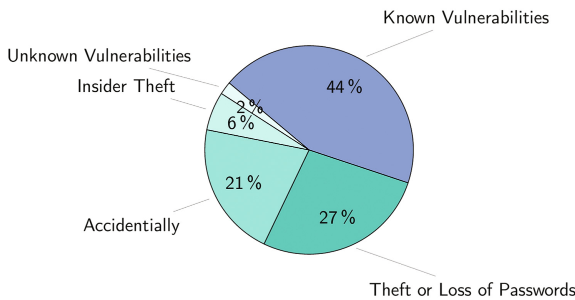

Fig. 1. Classification of attack causes

Especially due to the large number of attacks with known vulnerabilities, it is important to design new software in such a way that known vulnerabilities are no longer present. To ensure this, a software must be regularly tested against known vulnerabilities during the development cycle. This must include all known security gaps, because the old wisdom from project management is even more important in the field of security: "The later a problem is detected, the higher is the cost to fix it."

In the PC world, vulnerabilities are stored in a database of the MITRE Cooperation on behalf of U. S. Department of Homeland Security. This Common Vulnerabilities and Exposures Database [10] saves all known security gaps in existing applications. By the year 2016, about 100,000 attacks on various systems were recorded in this database. In addition, the MITRE Cooperation provides a database for the overview of all known

vulnerabilities in the Common Weakness Enumeration Database (CWE) [11]. There are currently about 1,000 different vulnerabilities in this database. In the CWE, the weak spots are divided into different categories. A classification of the attacks from the year 2015 leads to the distribution shown in Fig. 2. The most common attacks that are listed in the CWE are so called Denail of Service (DoS) attacks. These attacks make 33% of all known attacks on today's systems. The goal is to get the attacked system to crash and thus destroy the functionality of the system.

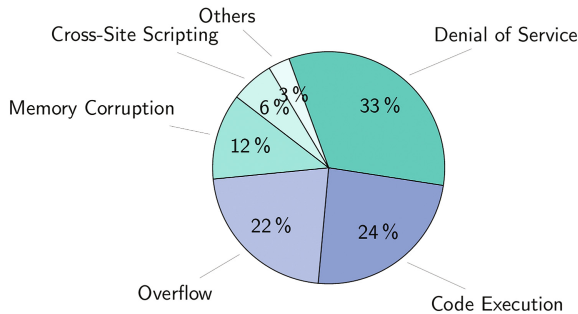

Fig. 2. Categorization of weaknesses according to CWE

More important than DoS attacks are attacks where an attacker can gain control over the entire system. Buffer overflows with 22% and code execution with 24% have a special significance. With so called buffer overflows (or overflows), memory areas are written with too long data sets to overwrite the following data records in the memory, thereby manipulating the contents of these variables. For an overview of attacks by buffer overflows, see Foster in [12].

The principle of buffer overflows is also used in code execution. Whereupon not only variables are overwritten, but the return address is set to a malicious, injected code of a function. This not only generates influence of the behavior by changing the variables but also take control over the entire system and execute malicious code.

Reason for the above attacks is usually a badly implemented software. Especially the consistent check of the memory area for dynamic variables can prevent overflow in most cases. However, due to time and memory space requirements in embedded systems, this is often omitted. One reason for DoS attacks is often the division by zero, which is not uniformly specified in microcontrollers and can therefore lead to different behavior or even program termination.

In addition to the above, there is often undefined behavior in software development when dereferencing so called null pointers that do not point to any memory, using memory or objects after executing "free", or read access to unallocated memory. In most cases, the aforementioned problems can be avoided by means of consistent queries in the programming, but the queries are rarely implemented for runtime and memory reasons.

Not all problems can be found by individual testing of the independent modules. Security weaknesses most often arise from the interaction of different modules and therefor the overall system needs to be tested as a whole.

4 Interception of Software Running on Virtual ECUs

Instruction set simulators like Open Virtual Platforms (OVP) [13] can be used to model a processor with the corresponding peripherals and run the cross-compiled application. Running the cross-compiled application inside an instruction set simulator gives the same behavior as on the target platform. The virtual prototyping of embedded systems for OVP is described by Werner in [14]. Werner also compares OVP with other platforms for virtual prototyping for embedded systems. He further explains in [15] the usage of OVP for debugging cross-compiled applications to build a virtual test environment.

The Imperas binary interception tool as defined in [16] can cause the simulation to stop the application and run the interception library at any point in time. This includes among others the interception of the virtual platform before each instruction is morphed, specific instructions are executed and a specific address range is read or written.

The interception technology is usually used for verification, analysis and profiling, including detection of memory corruption, deadlocks, data races or memory usage. As Imperas Software Limited writes in [16] this is especially useful when many different data scenarios should be executed.

With the binary interception tool we can use our own library to examine the state of the internal registers, instructions, memory, and other periphery. Furthermore, a replacement of the simulated behavior with a behavior defined in the interception library is possible. This means if the interception library detects a specific behavior during simulator the corresponding instruction is either replaced or extended by the one defined in the library.

The advantages of using the novel framework with interception libraries compared to other debug interfaces is that no additional code needs to be inserted in the application and no special access to the processor is needed. I.e. no resource overhead in the application and no additional instructions are executed. The application will be cross-compiled as running on the real hardware platform without any additionalities. Another advantage is that all parts of the interception technology will run in parallel to the simulation of the virtual platform.

As mentioned above we need to monitor the memory in order to find overflows and the instructions to find zero divisions. Both can be done by running an interception library in parallel to the main application.

An overview of the test framework can be found in Fig. 3. The platform for the virtual processor will be described in a platform model file as described by Werner in [14]. The virtual processor will consist of a processor with registers and memory for heap and stack, local memory for code and variables, as well as peripherals. The interception library will have direct access to these registers, memory, instructions, and peripherals. The location of the variables inside the registers and memory will be configured in a configuration file. Further, this file holds information about the intercepted instructions and functions. We are generating this file with information of the source files from the application. Therefore the source files are parsed and variables will be detected. The supervision of instructions will be done during run time with the disassembled application code, searching for divisions and illegal memory access.

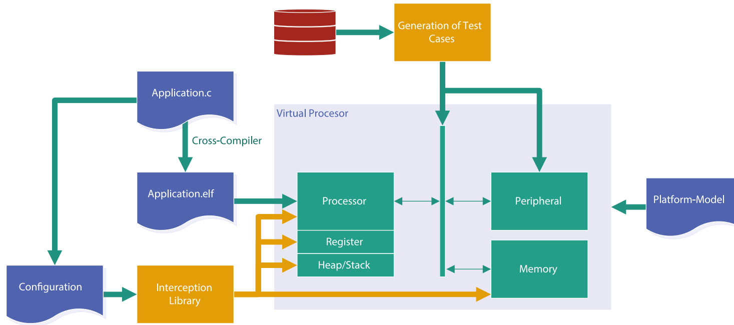

Fig. 3. Overview of security test-framework with virtual processor

4.1 Monitoring of Instructions

With the interception library [16] we can monitor all instructions on assembler level and check each of them if we need to observe the corresponding instruction. The behavior of the interception depends on the instruction. Our novel approach will intercept only potential vulnerabilities and directly executed all other instructions. E.g. we look at the different assembler instructions, if we find a division (either udiv for unsigned or sdiv for signed division) the corresponding registers will be checked for zero division. If the denominator is zero the execution of this command will be stopped and an error message will be displayed. If the instruction is not a division the interception library will not be executed.

To find potential vulnerabilities by zero divisions the observation of instructions can be implemented as static interceptions, because the instructions are well known at compile time and will be constant for all applications.

4.2 Monitoring of Memory Access

The same concept can be done with the memory. Each memory access (read and write) will be monitored and an error message will be displayed if data is written to the wrong memory range. The address range of the variables are stored in the interception configuration (see Fig. 3) and access to these ranges will be observed. If a write access across the variable boarders occur (buffer overflow) an error message will be displayed.

4.3 Heap and Stack Monitoring

For the heap and stack monitoring we need the memory tracing, because the local and dynamic variables will be stored at the local memory. Further we need the function tracing to trigger the interception whenever a function is called and new variables are stored on stack or heap.

The local variables will be pushed to the stack, therefore the instruction monitoring needs to add the variables to the dynamic monitoring, thus the interception library knows

the address and range of the new variables. The same will be done for dynamically allocated or deallocated memory inside the heap. This memory is usually allocated or deallocated with malloc and free. Another observer will find write access to the function return address to detect illegal code executions.

Both the dynamic memory observation and the observation of the return address needs to be done during runtime. Therefore a dynamic part of the interception library is necessary, that can be extended by the simulation.

5 Virtual Instrumentation for Security Testing

The security testing is based on the cross-compiled code for the target platform. I.e. the instructions order and the behavior is the same as on the real platform, since no further or different optimization of other compilers will be done. Further the Executable and Linkable Format (elf) file that is used for the testing can be flashed to the target device without any additional changes. Current state of the art tests (see above) are focusing on the source code without compiler optimization.

Even if the testing framework can check the source code using a static analysis before cross-compiling and running the application on the virtual processor, we are not focusing on this, since static code analysis is state of the art. This novel approach can even be used to run the compiled application without having access to the source code of the application. I.e. black box tests for security can be executed. But nevertheless information about functions and variables are needed in order to build the configuration file.

In the next step, after static code analysis, the application is checked for variables and functions. The static variables and functions will be added to the interception configuration. With this information the interception library is build and passed to the instruction set simulator. If the defined interceptions occur the simulator will stop the execution of the application and run the functions provided by the interception library.

The Imperas instruction set simulator is used to run the defined test cases and the interception library. For this step a model of the target platform is needed. This should include all necessary processors, memories, registers, and peripherals (see above). The interception library will stop the running cross-compiled application in the simulator at every predefined interception. Further if new memory is dynamically allocated the interception library will be extended to observe this memory area as well. After deallocation of the memory the corresponding entry in the interception library will be deleted.

At last the results of the simulation and the test process will be shown for documentation. The total workflow of the virtual instrumentation for security testing can be seen in Fig. 4.

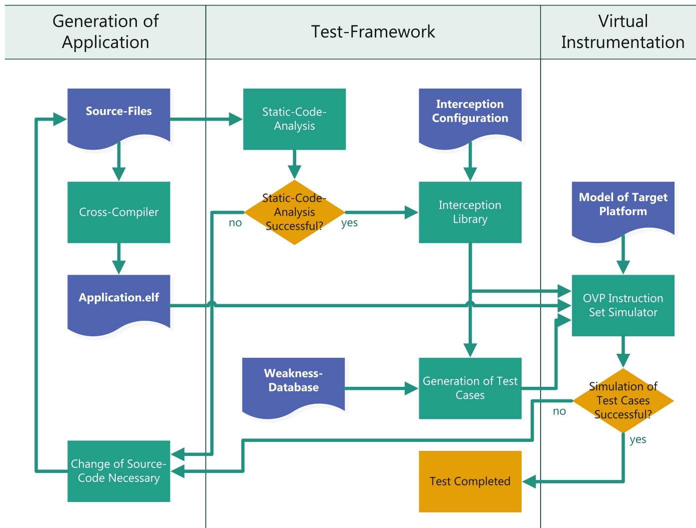

Fig. 4. Workflow of the virtual instrumentation for security testing

6 Conclusion and Future Work

In this paper we analyzed the different security weaknesses and derived from CWE that the most common ones are buffer overflow, code execution and division by zero. According to this knowledge we did a conceptual design for a security test framework based on virtual instrumentation. We build an interception library that monitors the memory and the instructions and reports security weaknesses if they occur.

Future work will investigate the concepts to automate the tests using virtual platforms. Further the memory observer for the variables and the test cases should be generated automatically. The interception library should be used to generate test cases according to the interfaces and a weakness database (CWE). These test cases will based on the information of variables (including dynamic variables) and functions from the application.

Another work will be done to use the framework for black-box-testing. This means without any knowledge of the source code. Especially the observation of dynamic variables have to be researched.

Protection of Shared memories and multi-processor systems can be tested as well. The virtual framework will be extended for the usage of multi-processor systems in the future.

Acknowledgement. This publication was written in the framework of the Profilregion Mobilitätssysteme Karlsruhe, which is funded by the Ministry of Science, Research and the Arts in Baden-Württemberg.

References

1. Kramer, J., Hillenbrand, M., Müller-Glaser, K.D., Sax, E.: Connected efficiency–a paradigm to evaluate energy efficiency in tactical vehicle-environments. In: Bargende, M., Reuss, H.C., Wiedemann, J. (eds.) 16. Internationales Stuttgarter Symposium. Proceedings, pp. 1451–1463. Springer, Wiesbaden (2016). doi:10.1007/978-3-658-13255-2_107
2. Koscher, K., et al.: Experimental security analysis of a modern automobile. In: IEEE Symposium on Security and Privacy, pp. 447–462 (2010)
3. Checkoway, S., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: USINEX Security Symposium (2011)
4. Bayer, S., Enderle, T., Oka, D.-K., Wolf, M.: Automotive security testing—the digital crash test. In: Langheim, J. (ed.) Energy Consumption and Autonomous Driving. LNM, pp. 13–22. Springer, Cham (2016). doi:10.1007/978-3-319-19818-7_2
5. Knechtel, H.: Methoden zur Umsetzung von Datensicherheit und Datenschutz im vernetzten Steuergerät. ATZ Elektronik **10**(1), 26–31 (2015)
6. Spillner, A., Linz, T.: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester; Foundation Level nach ISTQB-Standard, 4th edn. dpunkt.verlag (2010)
7. Radzkewycz, T.: Automotive networks can benefit from security. In: Connected Vehicle Journal: Designing for Next-Generation Connected and Autonomous Vehicles (2016)
8. Wheatley, M.: Known vulnerabilities cause 44 percent of all data breaches. http://siliconangle.com/blog/2016/01/12/known-vulnerabilities-cause-44-percent-of-all-data-breaches/. Accessed 31 Oct 2016
9. Symantec Corporation: Internet Security Threat Report. 2013 Trends, vol. 19 (2014)
10. MITRE Corporation: Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/. Accessed 31 Oct 2016
11. MITRE Corporation: Common Weakness Enumeration (CWE). https://cwe.mitre.org/. Accessed 31 Oct 2016
12. Foster, J.C., Osipov, V., Bhalla, N.: Buffer Overflow Attacks: Detect, Exploit, Prevent. Syngress Publishing Inc., Rockland (2005)
13. Imperas Software Limited: Open Virtual Platforms: The source of Fast Processor Models & Platforms. http://www.ovpworld.org/. Accessed 15 Dec 2016
14. Werner, S., et al.: Cloud-based design and virtual prototyping environment for embedded systems. Int. J. Online Eng. (IJOE) **12**(9), 52–60 (2016)
15. Werner, S., Lauber, A., Becker, J., Sax, E.: Cloud-based remote virtual prototyping platform for embedded control applications: cloud-based infrastructure for large-scale embedded hardware-related programming laboratories. In: Proceedings of 2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV). IEEE (2016)
16. Imperas Software Limited: Imperas Binary Interception Technology: User Guide, no. V1.5.3 (2016)