Securing the IoT

Utilizing conformance tessuites for fuzzing

Dorian Knoblauch
Software Quality Center (SQC)
Fraunhofer FOKUS
Berlin, Germany
dorian.knoblauch@fokus.fraunhofer.de

Sascha Hackel
Software Quality Center (SQC)
Fraunhofer FOKUS
Berlin, Germany
sascha.hackel@fokus.fraunhofer.de

***Abstract*— IoT devices are widely used in almost all vertical domains like homes, factories or as wearables on the body. This diversity is reflected in a variety of implementations which creates challenges for security testing due to the lag of applicability of out-of-the-box security testing solutions, like existing in other areas. We're introducing a security testing suite that is capable of providing security tests. Our security test suite is part of the Eclipse IoT-Testware project.**

It is capable of creating fuzz test cases from conformance test suites for devices automatically, regardless of the used protocols. Eclipse IoT-Testware reads into the communication between two devices, generates a model of the used protocol and generates fuzz data using the generation library Fuzzino. This solution has found vulnerabilities in ITS devices and flaws in devices using COAP and MQTT.

Keywords— IoT; negative testing; TTCN-3; Eclipse Titan; Fuzzing; security

I. INTRODUCTION

IoT devices are widely used in multiple areas like the private home, factories or even as wearables on the body. This diversity is reflected in a variety of implementations which creates challenges for security testing due to the lag of applicability of out-of-the-box security testing solutions, like existing for other areas.

Security testing is an important part of the test process, thus it reveals flaws in the applied security mechanisms and implementations in general. As systems getting more complex and interconnected, the attack surface increases. General security testing approaches are mainly focusing on discovery and avoidance of known vulnerabilities. This is quite effective for common software stacks and infrastructures but it fails as soon as new or proprietary technologies are subject to test, which very much applies to the domain of the Internet of Things.

Incidences and cyber-attacks, like the appearance of the Mirai-bot-Network [1], an attack on the Ukraine Power Grid or a German steel mill [2] [3], have shown the thread potential of insufficient secured IoT systems which nowadays, due to industry 4.0, are part of critical infrastructure.

Our approach introduces a security testing suite that's capable of providing suitable security tests according to the device's interfaces by listening to the communication of the IoT devices.

We take usage of conformance and interoperability test suites written in the Testing and Test Control Notation (TTCN-3)[1], a testing language standardized by the European Telecommunications Standards Institute (ETSI).TTCN-3 test suites usually test each functionality of a protocol do exits for all major communication protocols like SIP, WiMax, and IPv6. By taking those as the basis for our security testing we ensure that a high specification coverage is achieved even when dealing with a black box implementation. Our security test suite is part of the Eclipse IoT-Testware project, which includes conformance test suites for COAP and MQTT, which are the foundation for all of our security testing.

In the area of IoT, the most prominent TTCN-3 environment is the open-source tool Titan[2], whose procedures enable a fully automated testing. Tethering to those procedures, we developed a generic automated security test in TTCN-3 with the use of model-based fuzz-testing. Fuzz testing is a well-known, effective and widely accepted approach to identify and locate robustness and security-related weaknesses and vulnerabilities in a software-based system.

Fuzz testing is about systematically injecting invalid or unexpected input data to a system under test. That way, security-relevant vulnerabilities may be detected when the SUT processes such data instead of rejecting it. Eclipse Titan in its current version demands the definition of a new template with a precise data definition for each fuzzed message. In our approach, we'll ease this step up by providing fuzz data through the fuzz data generator Fuzzino to highly automate the test execution.

Fuzzino[3] is a library that supports the generation of test data for fuzz testing. It provides a set of data generation heuristics that target known weaknesses (e.g. integer or buffer overflows) and allows for finding new weaknesses by randomly modifying test data. In this, we'll show the implementation of the fuzzing approach, so about the difficulties on automatically map the TTCN-3 fields to the fuzz data.

1 http://www.ttcn-3.org

2 https://projects.eclipse.org/projects/tools.titan

3 https://github.com/fraunhoferfokus/Fuzzino

After providing the introduction of this paper in section I, a short discussion about related work follows in section II. In section III the decided test ecosystem including TTCN-3 as test language is explained. Section IV provides the implementation of the approach presented in this paper followed by the conclusion in section V. The paper finishes with an outlook in section VI.

II. Related Work

A. IoT

The Internet of Things (IoT) is a communicating-actuating network which exists due to the seamless dissemination into daily life blending information gathering and tasks executing devices [4]. IoT is driven by reused and newly emerged technologies and the networking architectures that are heavily utilizing the remarkable evolution that wireless and mobile communication have had. The challenge in such an interconnected environment is to obtain, in combination, a feasible high performance and security level [5].

Communication among vehicles is one major use case for the IoT and Intelligent Transport Systems (ITS) is one of the technologies pushing forward to make vehicle communication possible.

ITS are making an important and innovative contribution to more efficient, cleaner and safer mobility. Telematics and all types of communication, including technologies from the IoT domain, are enabling vehicles to exchange data among each other. Although a main focus of ITS is on wireless Short-Range Communications based on ad-hoc networks complementary data such as wide area traffic information is provided via the internet and conventional transmission technologies such as 5G or LTE. The standards are published by the ETSI and define the protocols and ports for communication [6]. As a proportionally undiscovered area in terms of security testing, ITS implementations are providing an ideal test ground for testing our approach.

B. Fuzzing

Fuzzing is an effective negative testing method of finding vulnerabilities in Software. In this black box approach, a system under test is exposed via its interfaces to unexpected data [7]. Typical interfaces are file loading mechanisms, network protocols and proprietary interfaces like API's [8]. The expectation is that due to the exposure with partially invalid data, the system gets into an unexpected state. Inputs of this type can either be generated randomly or systematically by mutating valid data or creating new data according to specifications [9]. The vast majority of inputs gets rejected by the system because of mechanisms like input validation, mapping to other data representations or elevation to an upper layer. Those rejected inputs are considered ineffective since their execution doesn't lead to the possibility of exposing a weakness which reduces the overall effectiveness of a fuzzing campaign. Model-based security testing does target this issue by generating test cases according to the systems model and specifications [10].

C. TTCN-3

The Testing and Test Control Notation (TTCN-3) is an international test language, standardized by the ETSI. Originally, it has been used for black-box conformance protocol testing in the telecommunication domain. Nowadays, the language widens its scope and is also suitable for performance or security testing in different domains like automotive, medicine, or banking.

The technology is explained very well in articles and books [11]. It has been accepted in several international projects, performed through e.g. SIP tests [12], 3GPP or the WiMax-Forum certification programs.

D. Tool support

Fuzzing is supported in all its various aspects by all kinds of tools. One that is currently well reflected in the media due to its success in finding vulnerabilities in a variety of different systems is the mutation based *American Fuzzy Lop* (afl). By recompiling the code with special flags, afl gains the opportunity to obtain information on code coverage. The goal of each fuzzing campaign is to produce inputs where its execution covers all paths in a program, which ultimately increases the overall possibility of determining a vulnerability. In this approach, we'll obtain a high coverage of the SUT implementation by generating test cases from existing test suites which by definition are meant to over a comprehensive feature set.

Just knowing the model doesn't naturally provide data that is increasing the likelihood of exploiting a weakness. The *Fuzzino* library fills this gap by generating data according to the model's needs via parameterization. For instance, taking an input field of a web application as a target, fuzzing would provide malicious strings or even SQL-injection string for the underlying database.

Eclipse Titan is an execution environment for the TTCN-3 language initially developed as an in-house tool by Ericsson. It supports the generation of executable and protocol independent test code out of TTCN-3. The target language is C++. Besides the core components, which includes amongst others the TTCN-3/ASN.1 compiler and a base library to support the code generation, it provides a graphical IDE based on the Eclipse IDE. To bridge the gap between generated code and the system under test (SUT), Titan uses test ports that are written ideally by domain experts with knowledge of the particular protocol or device.

We'll use the fuzzing language extension and data provided by Fuzzino to generate reasonable test cases that is described in section IV.A.

E. Integration of fuzzing into a TTCN-3 test setup

Like mentioned above, integrating fuzzing into an existing conformance test framework allows the utilization of model knowledge. In the particular case of TTCN-3 test frameworks and suites, package data unit descriptions are provided in the form of TTCN-3 templates, XSD-Definitions or ASN.1 notation and can be yield as models for the test case generation.

Already existing ports in TTCN-3 can be used for sending the generated fuzzing test cases.

A similar approach is described in *T3FAH: A TTCN-3 based fuzzer with attack heuristics*. This work of Luo et al. describes how to automatically extract the input syntax of the SUT from existing test data definitions in TTCN-3 conformance test suites. This syntax is used to generate invalid inputs based on their attack heuristic generation algorithm. In the final step, a fuzzing test script via reusing the conformance test case get automatically constructed [13].

Knoblauch and Großmann are describing an approach that is capable of learning each PDU model described in TTCN-3 and using this model to generate new test cases. This process is fully automated and needs just one example input to generate any amount of test cases using the model information and data provided by fuzzino library [14]. This approach was successfully evaluated and proved to be able to detect vulnerabilities in ITS implementations. The concrete implementation heavily relies upon a fully implemented TTCN-3 TRI [15] and the Java reflection API which is present in Spirent's TTWorkbench.

III. Approach

Today most projects have a lack of formal (i.e. machine processable) system model of the SUT available. Time pressure does not allow developing any formal model to get automatic tool support for test suite derivation. Thus, a manual written synthesis of the test suite is required. With this approach, it is difficult to reach high coverage of the functional requirements in affordable time. Even worse if requirements are related to security. To overcome this problem, we leverage the fact of a standardized test language such as TTCN-3, that is already used in several standardized test suites.

One of the main reasons for the success of TTCN-3 is the platform independence. A major step for the dissemination of the technology has been two years ago with the availability of the powerful Titan tool. Furthermore, TTCN-3 provides the flexibility to test different kinds of quality of a system. From traditional functional testing, the language concepts of TTCN-3 allow to include testing of non-functional requirements like security [14] [16], or performance [17] [18].

ETSI and other standardization bodies are issuing TTCN-3 test suites to achieve a high level of conformity and interoperability among different vendors. Mostly all important telecommunication protocols are covered with proper TTCN-3 test and currently, there are major efforts to provide similar test suites for IoT based protocols. One of it is the IoT-Testware project[4] at the Eclipse Foundation initiated by Fraunhofer FOKUS and several partners from industry and the research domain. The aim of the IoT-Testware is to supply a rich set of test suites for IoT technologies implemented with TTCN-3. Initially, IoT protocols MQTT and CoAP were in the focus. Later more technologies, like OPC UA, follows. It will help parties working with IoT relevant protocols or services to increase their quality by working collaboratively at the test suites within the project and make the results available at no charge in the scope of the Eclipse Foundation. Recently, next to conformance test suites, the IoT-Testware provides a yet small but promising ecosystem, including a full GUI, that was published recently.

Like mentioned in section II, those conformance test suites are covering huge parts of the functionality of the protocols and the interfaces. Ideally, most of the parts of an implementation of a standard, in a particular SUT which gets introduced to such a test suite, will be executed. This nearly full coverage which reflects in the TTCN-3 data description and the TTCN-3 communication procedures will be used to achieve similar high penetration with fuzz test cases. Already implemented mechanisms for sending data to the SUT like ports in TTCN-3 are being reused for sending fuzzed messages. The integration of an additional fuzz testing capability into an existing test suite showed to be the most reasonable approach

IV. Implementation

A. Test ecosystem

As described in II.D, Titan is used as the core platform to automatically generate test code and to embed the Fuzzino library as fuzzed data provider. In our approach, we're using existing parts of conformance and interoperability test frameworks to utilize them as models shown in Figure *I*. So modeling allows to narrow down the test cases and reduce the amount of data while keeping a similar effectiveness compared to a comprehensive approach. In terms of fuzzing capabilities, Titan introduces specific language extensions which currently are not in the TTCN-3 standard but are planned to be. It allows manipulating defined templates using the language specific `with` attribute in combination with the non-standardized `erroneous` keyword as described in detail in a post [19] at the official Eclipse Titan forum. It gives the possibility to update already existing test data rather than manually creating several new. Hence, we stick to the TTCN-3 templates taken from the conformance test suite and keep the same coverage. Because it is still not a standardized feature, we won't use this approach. However, this paper follows another approach of deriving fuzzed tests from conformance test suites manually and play them back into the Titan tool.

B. Implementation steps

1) Get the base

For applying fuzz testing we use conformance tests as a model. This allows us to obtain the structure of the message we send to the SUT. As we use existing tests as a base, we get the same coverage for the fuzz tests that was achieved with the conformance tests. It enables us to focus on the generated data we use as negative testing input for the protocol fields instead of asking for completeness. In this paper, we consider the ITS conformance test suite provided by ETSI as well as the IoT Testware for COAP and MQTT.

2) Building the model

4 https://projects.eclipse.org/projects/technology.iottestware

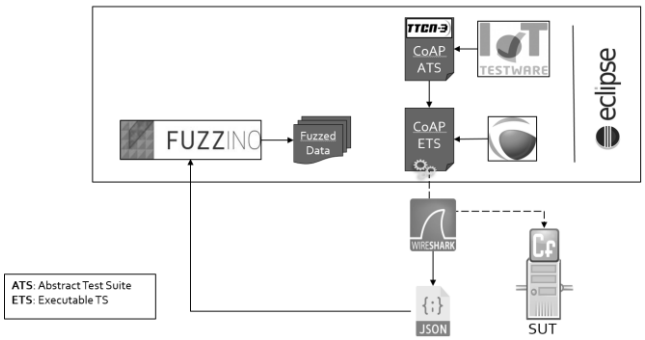

Figure I - Schematic overview of generating fuzzed data from already existing conformance tests.

The next step is to configure the conformance test suite from *1)* and execute it against a SUT. We record the produced traces with a packet analyzer, e.g. Wireshark. Thus, the traces reflect the packet structure that is already used by the conformance tests and will be re-used in the following steps. This step requires the packet analyzer to be capable of parsing the packages received by the SUT.

3) Generate fuzzed data

With the model by hand, we have an input for the Fuzzino library. The Fuzzino library provides a Java and XML interface to generate fuzzed data. The recorded traces from Wireshark are exported in a suitable exchange format. We decided to use JSON as Wireshark export because it is easy to process furthermore. This JSON representation contains the structure of the package/PDU including the precise bit allocation for each filed. Which is needed to modify the field data without tampering with structure. Wireshark does not provide the data type of the filed, therefore the initial allocation for the field get analyzed for its type.

Given size and type of the field, Fuzzino is capable of providing new data for each field. Such provided data is specifically generated for the purpose of revealing flaws. For instance, given a field with the type Integer and a length of 8-bit Fuzzino will provide amongst others values in boundaries, like 0 and 256. This gets more complex when dealing with Strings, here Fuzzino generates ranging from simple wrongly encoded String up to complex wrong formatted Http-requests or even SQL injections. This is shown in Figure *I*.

4) Narrow down the amount of test data

The allocation of the fields is key to the success of the fuzz test. Meaning that allocating more than one field with malicious unexpected data increase the possibility of causing unexpected behavior. Ideally, a fuzz campaign should make an n-wise allocation of each field in the package with newly generated fuzz data. But this will produce such a high number of test cases that one execution will likely run for months. Therefor we're using at max a pairwise allocation for each newly generated fuzzed package.

5) Run tests with fuzzed data

Having the initial PDU mutated by allocation chosen field with unexpected data created a new fuzzed PDU. This new PDU is being sent to SUT using the same communication stack as the conformance tests do. It occurs that sending the fuzzed PDU on the same layer might not be possible since its being altered to a degree where it fails to be encoded on the test system side. In this case, the fuzzed PDU has to be sent as a payload of the underlying layer. E.g. sending a fuzzed COAP message is not possible anymore on the provided TTCN-3-Port, since it raises a codec error, but sending a UDP payload will not introduce this issue.

Running the Fuzzing campaign means generating multiple fuzzed PDU for each in *2)* recorded PDU. It might be necessary to recalculate certain field of the PDU, like checksums or timestamps for decreasing the number of similar rejections by the SUT. Each fuzz PDU gets send to the SUT as shown in Figure *II*. At the same time, the reaction of the SUT gets monitored.

6) Log the results

Logging can be performed in different ways, but it's important that it includes the reaction of the SUT. At least it should be frequently checked if the SUT still responds. Those kinds of checks have to be implemented or at least configured based on each type of SUT. A cheap to implement version of such a check can be the execution of the regular conformance test suite after a certain amount of fuzz tests.

V. CONCLUSION

We've introduced security testing as an addition to the IoT-Testware. Our fuzzing approach of using a conformance test suite as the basis for fuzz testing ensures a high specification coverage. Automatic generation of fuzz test for each protocol that can be analyzed makes the application of security testing very easy. Our solution is capable of finding vulnerabilities as shown with ITS devices [14] and instabilities in COAP and MQTT implementations.

VI. OUTLOOK

With an introduction of a proper TRI implementation in Eclipse Titan, more sophisticated approaches like automated model exploration can be utilized to increase performance and reduce implementation effort.

Moreover, the results obtained in this paper can be made available for test suites of the IoT-Testware project. Because of the time-consuming execution of security tests with the help of the fuzzing approach, due to the big amount of generated data, could be a criterion for users to not use it. We suggest introduceing a lighter version of the fuzzing approach. That includes a reduced amount of fuzzed data that are fixed for the dedicated test suite.

Another field we are applying security tests is providing a basic test specification for IoT devices. This so-called

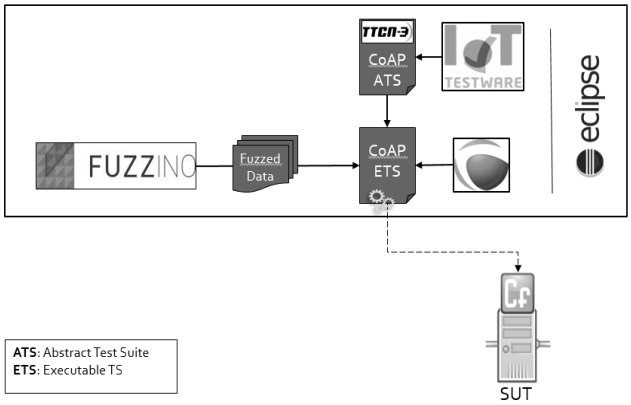

Figure II - Schematic overview of running the generated fuzzed data within the test ecosystem.

foundational security profile [20] defines a subset of security requirements from the IEC 62443 standard that represents the lowest security level, i.e. security level 1. With this set at hand, concrete formal test specification is written. Normally, standardization bodies or the industry don't provide those test specifications.

REFERENCES

[1] S. Gallagher, “How one rent-a-botnet army of cameras, DVRs caused Internet chaos”, online, accessed 17-august-2018

[2] K. Zetter, “Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid”, online, accessed 17-august-2018

[3] K. Zetter “A Cyberattack Has Caused Confirmed Physical Damage for the Second Time Ever”, online, accessed 17-august-2018

[4] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, “Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions”, Elsevier Science Publishers B. V., 2013, pp. 1645-1660

[5] H. Chaouchi, “The internet of things: connecting objects”, John Wiley & Sons, 2013

[6] ETSI, “Intelligent Transport Systems (ITS); Communications Architecture”, ETSI EN 302 665, 2010

[7] B. P. Miller, L. Fredriksen, B. So, “An empirical study of the reliability of UNIX utilities”, Communications of the ACM, vol. 33, pp. 32-44, 1990

[8] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, “SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr”, pp. 343-358, 2006

[9] A. Takanen, J. DeMott, C. Miller, “Fuzzing for Software Security Testing and Quality Assurance”, Artech House, Inc., 2008

[10] I. Schieferdecker, J. Großmann, M. Schneider, “Model-Based Security Testing”, Electronic Proceedings in Theoretical Computer Science, pp. 1-12, 2012

[11] C. Willcock, “An Introduction to TTCN-3”, Wiley, 2011

[12] A. Wiles, “Experiences of using TTCN-3 for Testing SIP and OSP”, 2001

[13] X. Luo, W. Ji, L. Chao, “T3FAH: A TTCN-3 based fuzzer with attack heuristics”, WRI World Congress on Computer Science and Information Engineering, pp. 744-749, 2009

[14] D. Knoblauch, J. Großmann, “Fuzz testing ITS”, UCAAT, 2016

[15] ETSI, “ETSI ES 201 873-5 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)”, pp. 1-320, 2015

[16] L. Zhou, X. Yin, Z. Wang, “Protocol Security Testing with SPIN and TTCN-3”, 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 511-519, 2011

[17] G. Ziegler, “Run-time test configurations for load testing”, 2007

[18] G. Ziegler, G. Réthy, “Performance testing with TTCN-3”, 2006

[19] E. Lelik, “Negative Testing in Eclipse Titan: Fuzzing basics.”, online, accessed 20-august-2018

[20] A. Wardaschka, A. Rennoch, S. Hackel, “Formalized Test Purposes for an Industrial Security Profile”, Proceedings 3rd GI/ACM Workshop , pp. 1-6, 2018