Security Testing by Telling TestStories

Michael Felderer[1], Berthold Agreiter[2], Ruth Breu[3] and Alvaro Armenteros[4]

Abstract: Security testing is very important to assure a certain level of reliability in a system. On the system level, security testing has to guarantee that security requirements such as confidentiality, integrity, authentication, authorization, availability and non-repudiation hold. In this paper, we present an approach to system level security testing of service oriented systems that evaluates security requirements. Our approach is based on the Telling TestStories methodology for model–driven system testing. After the elicitation of security requirements, we define a system and a test model. The test model is then transformed to executable test code. We show how traceability between all artifacts can be established and how the tests can be executed focusing on security relevant aspects. All steps are explained based on an industrial case study.

1 Introduction

While testing functional system requirements is one of the core software engineering disciplines, testing security requirements is a new emerging field. We contribute by defining and executing security tests based on the Telling TestStories (TTS) approach [FBCO+09], a methodology for model–driven system testing of *service oriented systems*. Service oriented systems consist of a set of independent components interacting via services to execute collaborative or managed processes. Based on a taxonomy of requirements, TTS defines a system model for a service oriented system and a related test model that invokes service operations of the system model. All model artifacts and the executable services are traceable. More details on TTS are presented in [FBCO+09, FFZ+09].

The work at hand shows how security requirements can be specified as functional requirements according to the TTS methodology such that tests of security requirements can be treated like functional system tests. Functional security tests, as defined in our approach, are more powerful than tests with other security testing approaches because our methodology extends the set of testable security requirements.

Based on the definition of security requirements we design a system model and test model containing test stories that are traceable to security requirements. Test stories are then transformed to executable test code which makes security requirements executable. We also show how the tests can be executed by integrating the test component as passive

[1] Institute for Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria, michael.felderer@uibk.ac.at
[2] Institute for Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria berthold.agreiter@uibk.ac.at
[3] Institute for Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria, ruth.breu@uibk.ac.at
[4] Telefónica I+D, C/ Emilio Vargas 6, 28043 Madrid, Spain, aap@tid.es

participant into the process under test. The methodology is demonstrated by an industrial case study.

The paper is structured as follows. In the next section, we present our case study and show how security requirements can be modelled and tested with TTS. In Section 3 we provide related work and finally in Section 4, we sum up and draw conclusions.

2 Security Modeling and Testing

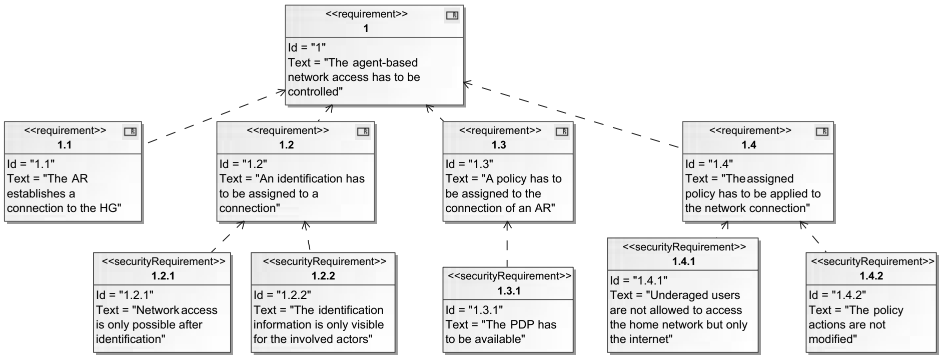
This section explains our security testing approach by TTS based on an industrial case study[5] to control the network access of clients in a home network. Depending on client properties such as the age of a user or the status of the installed anti–malware application, the network access control applies policies, e.g. that an underage user may only be allowed to access a restricted set of resources on the network.

The scenario consists of different peers distributed among the home network and the operator network. These peers are the *Access Requestor* (AR), *Home Gateway* (HG), *PolicyEnforcement Point* (PEP) and the *Policy Decision Point* (PDP). Following service oriented principles [Erl05], each peer shares interfaces defining the terms of information exchange. The AR is the client application to establish and use the connection to the home network. An AR always connects to a HG. The HG is a device installed at the home of customers controlling access to different networks and services (e.g. domotics, multimedia, data services). The enforcement of who is allowed to access which resources on the network is made by an internal component of the HG called PEP. The PEP gets the policy it has to enforce for a specific AR by the PDP which is the only component run by the operator and not the end user herself. Because we have four independent components only interacting via well-defined interfaces to execute a process, the example adheres to our definition of a service oriented system. Furthermore, we are focusing on testing dedicated example sequences (i.e. the test stories) of the system and verify whether certain security requirements hold under such conditions. The TTS framework adheres to a test-driven development approach, thus it allows the execution of test stories in early stages of system development and supports the evolution of the underlying system. Thus, TTS is an ideal choice for a testing framework of the presented system.

2.1 Requirements Model

The requirements are modelled by a requirements hierarchy. It defines a refinement from abstract requirements resp. goals to more detailed requirements. Security requirements and any other type of non–functional requirements can be integrated into this hierarchy in a natural way. For this purpose, we use SysML requirements diagrams [OMG07] and mark security requirements with the stereotype `securityRequirement`. Several classifications of security requirements can be found in the literature, e.g. [Fir03]. In this work we consider confidentiality, integrity, authentication, authorization, availability and nonrepudiation. In Figure 1 the security requirements relevant to our case study are represented in a requirements hierarchy.

[5] The case study was kindly provided by Telefónica.



<<requirement>> 1
Id = "1"
Text = "The agent-based network access has to be controlled"
<<requirement>> 1.1
Id = "1.1"
Text = "The AR establishes a connection to the HG"
<<requirement>> 1.2
Id = "1.2"
Text = "An identification has to be assigned to a connection"
<<requirement>> 1.3
Id = "1.3"
Text = "A policy has to be assigned to the connection of an AR"
<<requirement>> 1.4
Id = "1.4"
Text = "The assigned policy has to be applied to the network connection"
<<securityRequirement>> 1.2.1
Id = "1.2.1"
Text = "Network access is only possible after identification"
<<securityRequirement>> 1.2.2
Id = "1.2.2"
Text = "The identification information is only visible for the involved actors"
<<securityRequirement>> 1.3.1
Id = "1.3.1"
Text = "The PDP has to be available"
<<securityRequirement>> 1.4.1
Id = "1.4.1"
Text = "Underaged users are not allowed to access the home network but only the internet"
<<securityRequirement>> 1.4.2
Id = "1.4.2"
Text = "The policy actions are not modified"

Fig. 1: Requirements

In this representation, to each security requirement as to all other requirements, model elements of the test model (test stories, assertions, test sequence elements) verifying the requirement can be assigned. Normally, compliance to a security requirement will be checked by one or more assertions. In our basic requirements hierarchy, security requirements are formulated as positive statements, defining how a vulnerability can be avoided. Attached test stories may then define possible vulnerabilities that make the requirement fail. In Figure 1 we have defined an example for different types of security requirements. Requirement 1.2.1 is an example for authentication, 1.2.2 for confidentiality, 1.3.1 for availability, 1.4.1 for authorization, 1.4.2 for integrity and 2.1 for non–repudiation.

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction [GF94]. Traceability has to be guaranteed by a system testing approach to report the status of the system requirements. Our representation of requirements allows for a traceability definition by links between model elements, i.e. by assigning test stories to requirements. Because service operation calls in test stories are linked to service operation calls in the system model which are linked to executable service operations in the system implementation, we have traceability between the requirements model, system model, test model and the executable system.

2.2 System Model

As already mentioned, the AR is the client application to establish and use the connection to the home network. We model this with an `AccessRequest` interface required by the AR. This interface is provided by the HG because an AR always connects to a HG. The data used to decide to which networks a client is granted access is retrieved via the `Identification` interface which is provided by the AR. The HG uses an internal component to enforce these restrictions, the PEP. The PEP receives the policy it has to enforce for a specific AR by the PDP via the `PolicyDecision` interface.

All components in this scenario are connected with each other and the interfaces between them are well-defined (see Figure 2). A request by the AR will trigger the input of user credentials via the identification interface. The data returned by the AR is of type

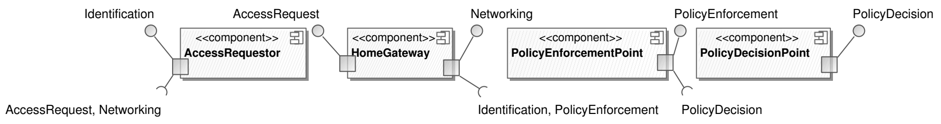

Identification
AccessRequest
Networking
PolicyEnforcement
PolicyDecision
<<component>> AccessRequestor
<<component>> HomeGateway
<<component>> PolicyEnforcementPoint
<<component>> PolicyDecisionPoint
AccessRequest, Networking
Identification, PolicyEnforcement
PolicyDecision

Fig. 2: Actors modelled as components with provided and required interfaces.

`IdentificationData` (see Figure 3). With this information the PDP is able to look up the appropriate policy for the request and send the corresponding list of `PolicyActions` to the the PEP which enforces them. `PolicyAction` and `IdentificationData` are data types defined internally in the system model. The type `IdentificationData` describes a username, a password and optionally attestation data of an AR; the type `PolicyAction` is currently only used to describe to which VLAN the PEP should allow access by a specific AR.

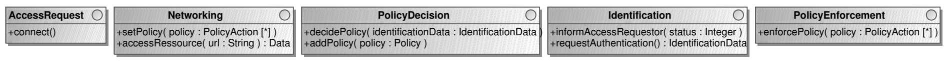

AccessRequest
+connect()
Networking
+setPolicy(policy : PolicyAction [*])
+accessRessource(url : String) : Data
PolicyDecision
+decidePolicy(identificationData : IdentificationData)
+addPolicy(policy : Policy)
Identification
+informAccessRequestor(status : Integer)
+requestAuthentication() : IdentificationData
PolicyEnforcement
+enforcePolicy(policy : PolicyAction [*])

Fig. 3: Interface definitions of services.

The communication among the peers is based on different protocols and standards. Authentication follows IEEE 802.1X[6] which defines a supplicant, an authenticator and an authentication server. In our case the supplicant is the AR, the role of the authenticator is taken over by the HG and the authentication server (a RADIUS[7] database) is represented by the PDP. Note that the system model describes all components, their interfaces and optionally also behavioural parts of the system. For the present contribution we only show the components and interface definitions as it suffices to describe the present scenario.

2.3 Test Model

The TTS test model contains a set of test stories whose execution order is defined in a test sequence. To make all requirements executable, we assign an assertion, a test story or a whole sequence element to it. Due to space limitations we present just one complex and representative test story in Figure 4 and show how the requirements can be mapped to it.

The test story in Figure 4 defines a basic network access scenario containing two assertions. First the `AccessRequestor` connects to the `HomeGateway` (step 1 in Figure 4), which then requests the authentication data containing a username, a password and assessment data from the `AccessRequestor` (steps 2 and 3). This information is forwarded to the `PolicyDecisionPoint` (step 4), which sends a sequence of policy actions to the `HomeGateway` (step 5) based on the identity information of the `AccessRequestor`. We then assert that there has to be a policy action that contains the expected VLAN (`$vlan`) to check the policy actions for integrity. The `HomeGateway` sends the policy actions to the `PolicyEnforcementPoint` (step 6), and then informs the `AccessRequestor` (step 7), which then accesses a specific URL (steps 8 and 9). Finally, we check whether the

[6] Available at `http://www.ieee802.org/1/pages/802.1x-rev.html`.
[7] Remote Authentication Dial In User Service, as specified in RFC 2865.

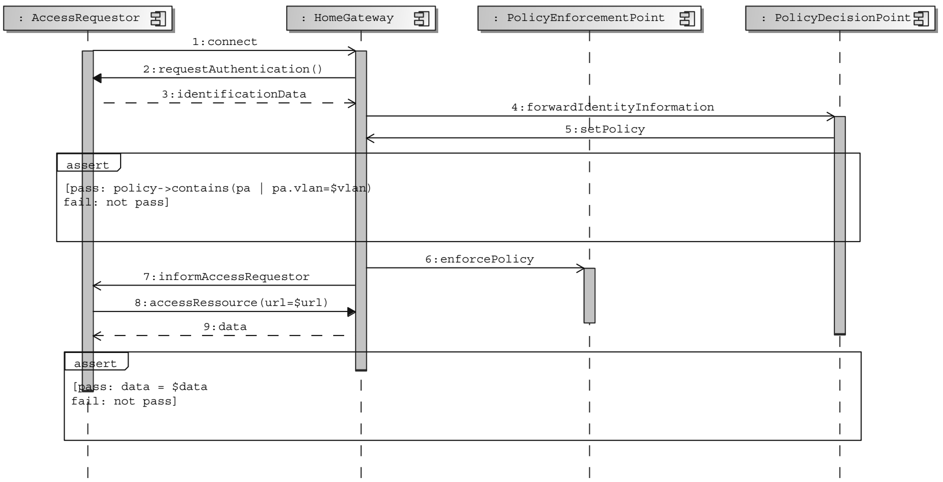


Fig. 4: Test story `TestPolicy`

`accessRessource()` call returns the expected data. Test cases for this test scenario are defined in Table 1.

`#TC`	`$username`	`$password`	`$vlan`	`$url`	`$data`
1	'michael'	'0815'	'HomeNetwork'	'http://74.125.43.99'	webpage_1
2	'michael'	'0815'	'HomeNetwork'	'http://192.168.1.1'	webpage_2
3	'philipp'	'0000'	'Internet'	'http://74.125.43.99'	webpage_1
4	'philipp'	'0000'	'Internet'	'http://192.168.1.1'	null
4	'guest'	'0000'	'Internet'	'http://192.168.1.1'	null

Tab. 1: Test Data Table

The test story is completed by adding some policies to the `PolicyDecisionPoint` in an initial setup. In our case, three policies are added to the `PolicyDecisionPoint`. Each policy assigns a sequence of policy actions, in our basic example just a list set of accessible VLANs, to a username/password combination. The identification data objects are stored in a data pool and are as follows in our example:

```
policy1:('michael','0815',(['Internet','HomeNetwork'])
policy2:('philipp','0000',(['Internet'])
policy3:('*','*',(['GuestNetwork'])
```

This initialization has to be executed before the test story `TestPolicy` can be executed for every test case of Table 1. Test sequence elements can contain additional arbitrations that aggregate the verdicts of the stories' test cases, e.g. such an arbitration could be `pass%=100%`, i.e. all test cases of a test story have to pass. The two assertions in our test story are traceable to requirements. In the requirements model of Figure 1, the first assertion can be assigned to Requirement 1.4.2 testing integrity, and Requirement 1.4.1 testing authorization. Additionally the overall test story can be mapped to Requirement 1.4 which is done implicitly in this case because the test story covers all sub requirements. Test stories, their states, test sequence elements and traceability for testing other requirements are similar to the one presented in Figure 4 but differ at least in the assertions.

2.4 Test Execution

The case study consists of four different actors communicating with each other. A crucial point of our testing strategy is that the different services are not tested individually and in an isolated way. Instead we define test stories which describe possible sequences of service invocations on the SUT. Testing each service separately is out of scope of our testing strategy. What we are interested in, is the value of certain parameters at specific points in a test story to evaluate assertions.

Another important point of our test execution technique is that the test engine is primarily a passive participant in this process. However, this is not a limitation of the Telling TestStories framework itself, see [FBCO$^+$09]. The reason for a passive execution engine lies in the scenario itself: all actors except the AR are hard-wired to each other. For instance, when the AR sends the `EAP-Response/Identity` message (i.e. the return value of the `requestAuthentication()`-call) to the HG this will trigger a message exchange between HG and PDP. Thus, a central execution engine acting as an orchestration unit is not reasonable in this scenario because it would simply “miss” certain messages. The test execution technique for this scenario starts a test story and only interacts with the `AccessRequestor`. The parameters for this interaction are given in the data table. The remaining communication is only observed by the execution engine. For monitoring this communication we use packet sniffers (TShark[8]) at various points in the environment so that we are able to track the full communication in a non-intrusive way.

Before the test story is started, the system is first set to a specific state. In our case the setup consists of a number of `addPolicy()`-calls to the PDP for installing the policies. After the system is initialized, the execution engine triggers the `connect()`-call by the AR. In the `requestAuthentication()`-call, the HG then requests the user credentials which are provided by the execution engine delivering values for the variables `$username` and `$password` from the data table. The next step involving the AR is the `informAccessRequestor()`-call where the AR is notified about the decision by the PDP. Immediately after this notification the AR can try to access a specific network resource via the `accessRessource()`-call. Again, the parameter for the requested URL is fetched from the data table, i.e. `$url`. The rest of the communication, where the AR is not involved, is only observed.

By monitoring all messages, the execution engine is able to keep track of the current value of variables defined in the interfaces among services, e.g. which `PolicyActions` are returned by the PDP. This information and the content of the data table are sufficient to compose assertions and to check the behaviour of the system. For example, the assertion `[pass: data = $data]` in the test story `TestPolicy` depicted in Figure 4 checks whether accessing a specific URL is allowed/denied as specified in the policy. This assertion can be evaluated by getting the value for `data` from the monitored return value of `accessRessource()` and the value for `$data` from the data table.

[8] Available at `http://www.wireshark.org`.

For each captured message of a running test story, the sniffer matches it to an interaction step of the test model and assigns the values according to the defined interface. After the test execution, the results can be evaluated.

3 Related Work

The topic of model-based testing is well-covered in the literature (see [BJK$^+$05] for an overview) and many tools are already on the market supporting model-based approaches, e.g. [UL06]. However, to the best of our knowledge, the contribution at hand is the first to combine model-based tests on system level with *security functional testing* and *security requirements testing* (cf. [Bis03]).

The aim of functional security tests is mainly the quality assessment of specified (security) requirements. In [JMT08, MJP$^+$07] the authors describe a model-based testing approach for checking whether access control policies are properly enforced by the SUT. The functional model is written in the B language and used for the security test generation from so called *test purposes*. Test purposes are defined as regular expressions and describe a general sequence of operation calls to induce a certain situation on the SUT. The approach aims at the automatic generation of test cases from the SUT. Our approach differs in several points: it supports test-driven development which also implies that test cases are not meant to be generated automatically but modelled by a domain expert (e.g. the customer); for the same reason, TTS allows for the execution of tests during system development. Our approach, furthermore, describes how the information passed among components is to be *interpreted* and how this information can be used to *check for compliance* with arbitrary security requirements and not only access control rules.

Vulnerability scanners, e.g. Nessus[9] constitute a tool-based approach to perform security tests on a very low level. Furthermore, only known vulnerabilities are detectable with such tools.

Other approaches, such as [WJ02], also use the system specification for generating security tests. However, our goal is not the automatic test case generation but a continuous connection among security requirements, tests and the system. Opposed to other security testing approaches, TTS supports a test-driven way of system development. This means that the system model does not have to be complete beforehand.

4 Conclusions

We have presented an approach to security requirements testing of service oriented systems by the methodology of Telling TestStories. Based on an industrial case study, we have defined a requirements model including security requirements, a static system model containing services, a test model and its execution. The test execution itself is accomplished by integrating the test engine as passive component into the process under test. Security requirements are specified as functional requirements. This enables the application of the TTS framework and extends the set of testable security requirements on the system level. We also guarantee traceability between security requirements and the executable system.

[9] Available at `http://www.nessus.org`.

Security testing which is generally considered as very difficult can be addressed by TTS in a clear, structured and intuitive way.

Our next step will be the implementation of our testing methodology on the real industrial system to conduct empirical data and to evaluate the system.

Acknowledgements. This work was partially supported by the SecureChange (ICT-FET-231101) EU project[10], the Telling TestStories project[11] funded by the trans-it and the MATE project funded by the FWF.

References

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley Professional, 2003.

[BJK+05] M. Broy, B. Jonsson, J.P. Katoen, M. Leucker, and A. Pretschner. Model-based Testing of Reactive Systems, volume 3472 of Lecture Notes in Computer Science, 2005.

[Erl05] T. Erl. *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[FBCO+09] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model-based Requirements Testing of Service Oriented Systems. In *Proceedings of the IASTED International Conference*, volume 642, 2009.

[FFZ+09] M. Felderer, F. Fiedler, P. Zech, , and R. Breu. Flexible Test Code Generation for Service Oriented Systems. 2009. QSIC'2009.

[Fir03] D.G. Firesmith. Engineering Security Requirements. *Journal of Object Technology*, 2(1), 2003.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. 1994.

[JMT08] Jacques Julliand, Pierre-Alain Masson, and Regis Tissot. Generating security tests in addition to functional tests. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, New York, NY, USA, 2008. ACM.

[MJP+07] P.A. Masson, J. Julliand, J.C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model based tests for a class of security properties. In *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM, 2007.

[OMG07] OMG. *OMG Systems Modeling Language*, 2007. http://www.omg.org/docs/formal/2008-11-01.pdf.

[UL06] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006.

[WJ02] G. Wimmel and J. Jürjens. Specification-based test generation for security-critical systems using mutations. *Lecture notes in computer science*, pages 471–482, 2002.

[10] Information available at `http://www.securechange.eu/`

[11] Information available at `http://teststories.info/`