D45.2 – Reference test environment

Server Testbed

Document Identification	
Date	06/12/2013
Status	Final
Version	1.1

Related SP / WP	SP4 / WP45	**Document Reference**	https://dms-prext.fraunhofer.de/livelink/livelink.exe/overview/3012742
Related Deliverable(s)		**Dissemination Level**	PU
Lead Participant	USTUTT	**Lead Author**	Eray Özmü
Contributors	Christopher Ruff, Tobias Wich	**Reviewers**	NRS

This document is issued within the frame and for the purpose of the *FutureID* project. This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424

Document Information

Contributors

Name	Partner
Eray Özmü	USTUTT
Christopher Ruff	USTUTT
Tobias Wich	ECS

History

Version	Date	Author	Changes
0.1	17.06.2013	Eray Özmü	Created Outline
0.2	02.08.2013	Eray Özmü, Christopher Ruff	Contribution on the wiki
0.5	10.09.2013	Christopher Ruff	Added Challenges, Conlcusion, etc.
0.6	20.9.2013	Tobias Wich	Contribution to Jenkins
0.7	25.09.2013	Eray Özmü, Christopher Ruff	Contribution to Jenkins and Test vectors General improvements
1.0	14.10.2013	Eray Özmü	Considered input from Review of NRS
1.1	06.12.2013	Eray Özmü	Corrected naming of components

Table of Contents

Abstract

FutureID with its locally distributed team builds a server infrastructure which will integrate with different identity providers and service providers and furthermore integrate client software to provide secure and user-friendly authentication. To be able to ensure thorough testing we implemented a first internal test environment and defined the test scope and testing process.

The testing process aims on the traceability of requirements and their respective test coverage. For additional testing the tasks t41.5, t42.7, t43.5 and t44.5 have been defined in the description of work document. These tasks will use the testbed which is being created in work package W45.

For the test environment we use three different tools. The first and central tool is called Jenkins and provides the control of all testing activities. Different jobs like pulling, building, testing and analyzing run automatically on a timely basis and ensure a continuous overview on the quality of the project.

SoapUI and Sikuli are used to test certain functions of the system. Whereas SoapUI concentrates on running tests for certain functions of the components, Sikuli will be used to test on the system level and run real use cases on the actual execution environments automatically.

1. Introduction

The *FutureID* project aims to build a sophisticated server infrastructure to integrate with the *FutureID* client in order to ensure secure, privacy-friendly and usable authentication with all different kinds of tokens. The development team works in a decentralized and geographically separate manner. To be able to ensure a high quality of the server infrastructure a server testbed needs to be created. This server testbed will help the developers and testers of the project to evaluate the developed software and define its quality.

This document states the current status of the first internal server test environment. The first internal test environment is built to help developers test the developed software accordingly. This document will not only concentrate on the tools which will be provided for testing, also the documentation of testing and respective requirements will be discussed.

In chapter 2 the testing process will be presented. The documentation of the test cases will be built upon the test assertion guidelines by OASIS [1]. Furthermore tools will be introduced which will help the project team throughout the project to keep track on all requirements and their testing stages. In chapter 3 the test environment itself will be presented. This includes the central continuous integration tool Jenkins [2] and two other tools for the actual testing tasks.

1.1 Scope and Limitations

Figure 1 shows the high level architecture of the *FutureID* client-server infrastructure for an updated and more complete version of this figure please look in deliverable D21.4. As this deliverable is only concerned with the server testbed, only the part within the red rectangle will be discussed. For the testing of the *FutureID* client please have a look at work package 37.

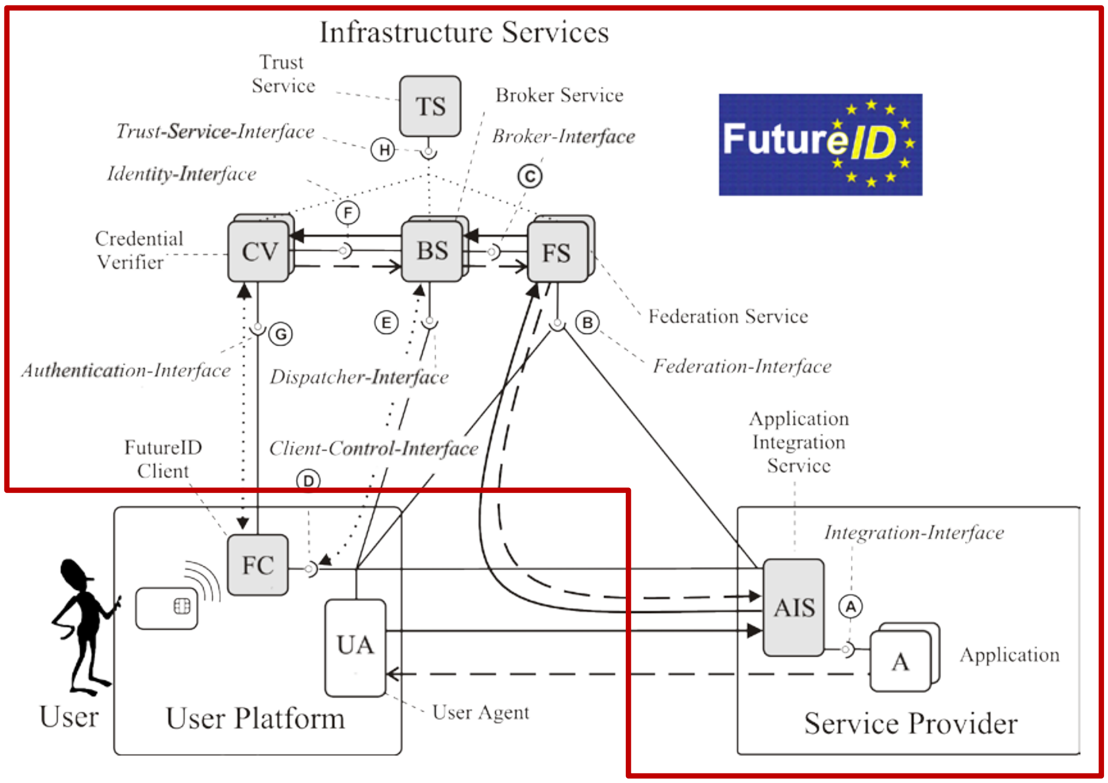


Figure 1 - *FutureID* Architecture

The high level goal of the server testbed is to provide the right toolset for the developers and testers of this project to fulfill thorough testing and to evaluate if the software meets the requirements defined by the different stakeholders. It is not intended to provide testing of the whole infrastructure within this work package. Instead this task will provide the first internal test environment.

1.2 Challenges

When testing a distributed system such as the Future ID server infrastructure there are unique challenges involved. There is a number of components that have to be set up and working in conjunction to allow testing the communication between them. Having a reliable reference test environment is crucial for a successful implementation of the server functionality. It is important to test the functionality from as many angles as possible and having the greatest possible test coverage of code and functionality. But as coverage can never be fully complete, the tests should focus on key functionality and consider the typical use-cases as well as degenerate ones. The communication between components has to be specified in detail to be able to test against this specification. Corrupted information in the specified communication protocols need to be tested as well as valid information to avoid erratic behaviour and minimize attack vectors. Testing should consider all levels from a business logic level to a low-level technical level. One the one hand, the business logic and functionality has to work as intended which will be ensured by testing on the various levels (see also section 2.2) on the other hand, there needs to be a strong focus on security testing as *FutureID* and its applications are intended for authentication and identity management services where security and integrity are essential.

2. Testing Process

The testing process can be divided into two sections. The first section will handle the documentation and traceability of requirements and the corresponding test cases. The second section will handle the testing process itself, as there will be different testing activities on different testing levels involved during test of the server infrastructure.

2.1 Traceability of Requirements

In large projects with the involvement of different partners from different countries it is difficult to maintain an up to date overview of all requirements during the project and keep track if the main must requirements are being tested and fulfilled.

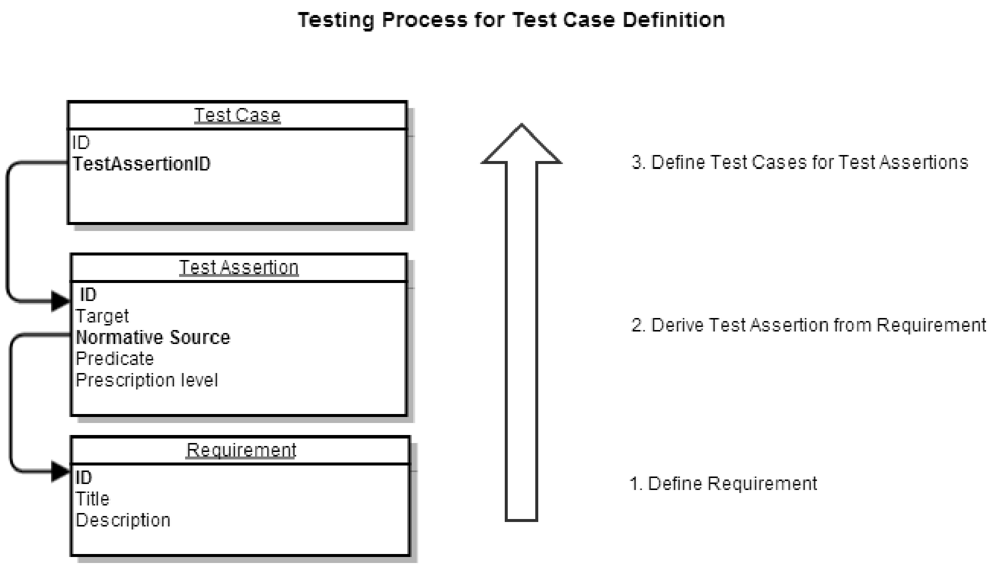


Figure 2 - Relationship between requirements, test assertions and test cases

The OASIS Committee released a guideline for the usage of so called test assertions. These test assertion are used to help raise the quality and understand specifications of software more deeply. It is a testable expression of a certain requirement. Based on this test assertion test cases can be derived. In Figure 2 the relationship between requirements, test assertions and test cases is pointed out. By cross-referencing, it will be easily comprehensible if and how certain requirements are tested.

It is not mandatory to always use test assertions, but in some cases it is very helpful to translate requirements into testable and measurable assertions. In other cases it is also applicable to directly derive test cases from certain requirements.

We will introduce a toolset for the definition of requirements and their respective test assertions and test cases. These will all be created in the semantic project wiki and can later on be exported in the XML format for further analysing. This will help to enable the systematic traceability of the requirements and their current testing stage.

2.2 Testing levels

Different functions and requirements must be tested on different levels. Whereas the communication between all modules should be tested within a system test, the specific modules must be tested differently. As a result there will be two different approaches on testing of the server architecture. The first approach is based on SoapUI and will be used to test specific methods and procedures. The second approach is based on Sikuli which is used to test the *FutureID* project on a system level by execution of real use cases and the evaluation of the project as a whole.

SoapUI [4] will be used to test specific modules with their “inside” behavior. One example for such a test is the authenticate function of the broker service which has been defined in deliverable D41.2. With SoapUI it is possible to run different authentication requests towards the broker service and automatically analyze their responses. These tests will be defined once and can then run on a timely basis. The focus will lay on certain functions and interfaces of each module.

As the testbed will be used by respective partners to test the components within the testing tasks (T.41.5, T.42.7, T.43.5 and T44.5) the descriptions of these tasks have been analyzed thoroughly. The following chapters provide the description of work of the tasks. For each task the most important parts for the internal testbed are highlighted within the text.

2.2.1 Task T41.5 - Integration and testing of Broker service

This task will integrate the Broker service with other components of the *FutureID* infrastructure (Client, Application Integration Services, Universal Authentication Service, Trust Repository and external authentication and identity services etc.) and will perform appropriate tests to guarantee the required level of quality as specified in D41.2.

This task will enable and assess the readiness of the Broker service to connect to STORK in dispatcher mode.

Participants: ECS (3 PM), ATOS (8 PM), CA (4 PM), SK (2 PM), USTUTT (2 PM), TUG (2 PM)

2.2.2 Task T42.7 - Integration and testing of Universal Authentication Service

This task will integrate the different parts of the Universal Authentication Service and will perform appropriate tests to guarantee the required level of quality as specified in D42.1. In particular the various authentication protocols described by APS-files are tested in conjunction with the execution environment D 42.7 and the basic services D42.5.

Participants: ECS (3 PM), CA (3 PM), ULD (1 PM)

2.2.3 Task T43.5 – Integration / Testing (Trust services module)

This task will integrate the Trust Services module into the *FutureID* backend framework and perform operational tests in order to validate the solution.

Participants: SK (2 PM), USTUTT (2 PM), TUG (3 PM), ULD (1 PM)

2.2.4 Task T44.5 – Integration / Testing

This task will integrate the application integration service implemented in Task 44.4 as a module into the *FutureID* backend framework and perform operational tests in order to validate the solution.

Participants: AG (3 PM), ULD (1 PM)

3. Test Environment

One main objective of this deliverable is the definition and setup of the first internal test environment for the server components of the *FutureID* project. The requirements of deliverable D45.1 have been analysed and used to define the right toolset and environment to help testers evaluate the server components easily and thoroughly.

The main platform for testing is Jenkins which will be used to gather and control all testing activities within one platform. The Jenkins platform is documented and explained in chapter 3.1. The main tool for test case implementation and execution is SoapUI, which will also be explained in detail in chapter 3.2.

The use of SoapUI with the combination of Jenkins won't be enough to thoroughly test all functions and components of the server. There are scenarios in which the client might misbehave and send requests to the server which are not predictable. In order to ensure that the server will always act accordingly and won't lead to the crash of the client or other exceptions, system tests are needed. Whereas system tests usually target the client application, it is mandatory to also use system tests to test the whole infrastructure together with the client software under real execution conditions. The tools for system tests are described in detail in chapter 3.3.

3.1 Jenkins

Jenkins is a continuous integration platform which is used to control all testing activities and to present the test results. For confidentiality reasons there will be a separate instance of Jenkins for the server components, as these components are not open source and need to be secured against code leakage. The Jenkins instance can be accessed at https://ci.FutureID.eu.

Interested parties should apply for an account to get access to Jenkins.

3.1.1 Dashboard Overview

The first thing that is visible to the user after visiting the service is the Dashboard as shown in Figure 3. It contains a coarse overview of the build configuration, status information, and trends about the last builds. Besides this information the individual widgets on the Dashboard contain references to views with more details and control possibilities.

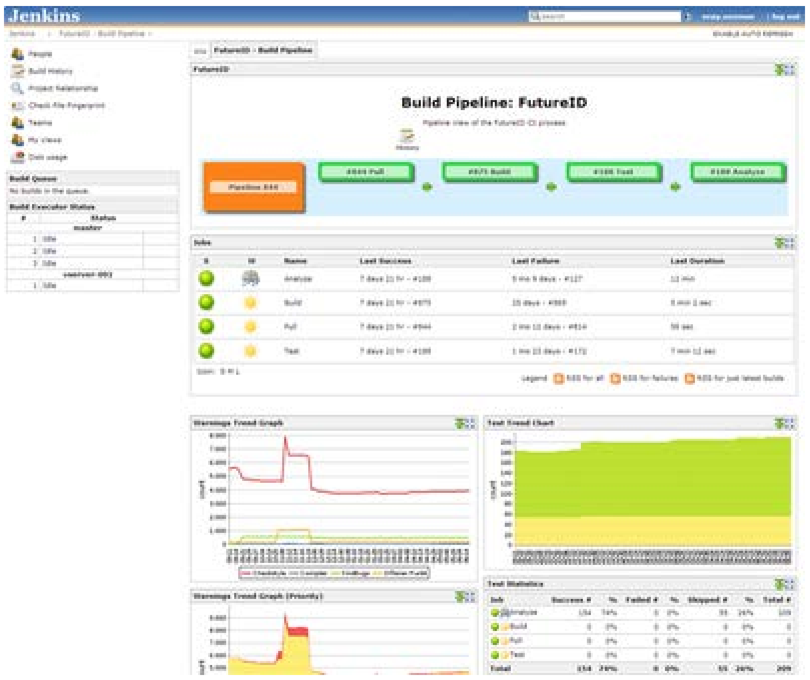
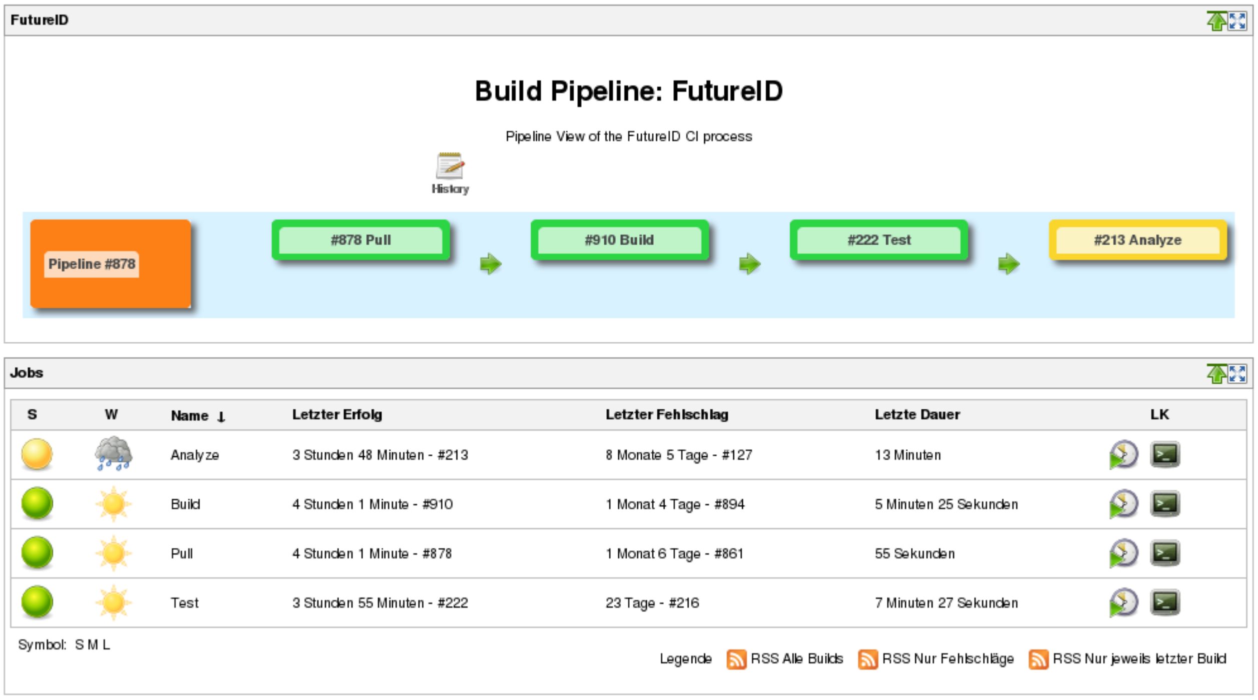


Figure 3 - Jenkins Dashboard


FutureID

Build Pipeline: FutureID

Pipeline View of the FutureID CI process

History

Pipeline #878 → #878 Pull → #910 Build → #222 Test → #213 Analyze

Jobs

S	W	Name ↓	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer	LK
		Analyze	3 Stunden 48 Minuten - #213	8 Monate 5 Tage - #127	13 Minuten	
		Build	4 Stunden 1 Minute - #910	1 Monat 4 Tage - #894	5 Minuten 25 Sekunden	
		Pull	4 Stunden 1 Minute - #878	1 Monat 6 Tage - #861	55 Sekunden	
		Test	3 Stunden 55 Minuten - #222	23 Tage - #216	7 Minuten 27 Sekunden	

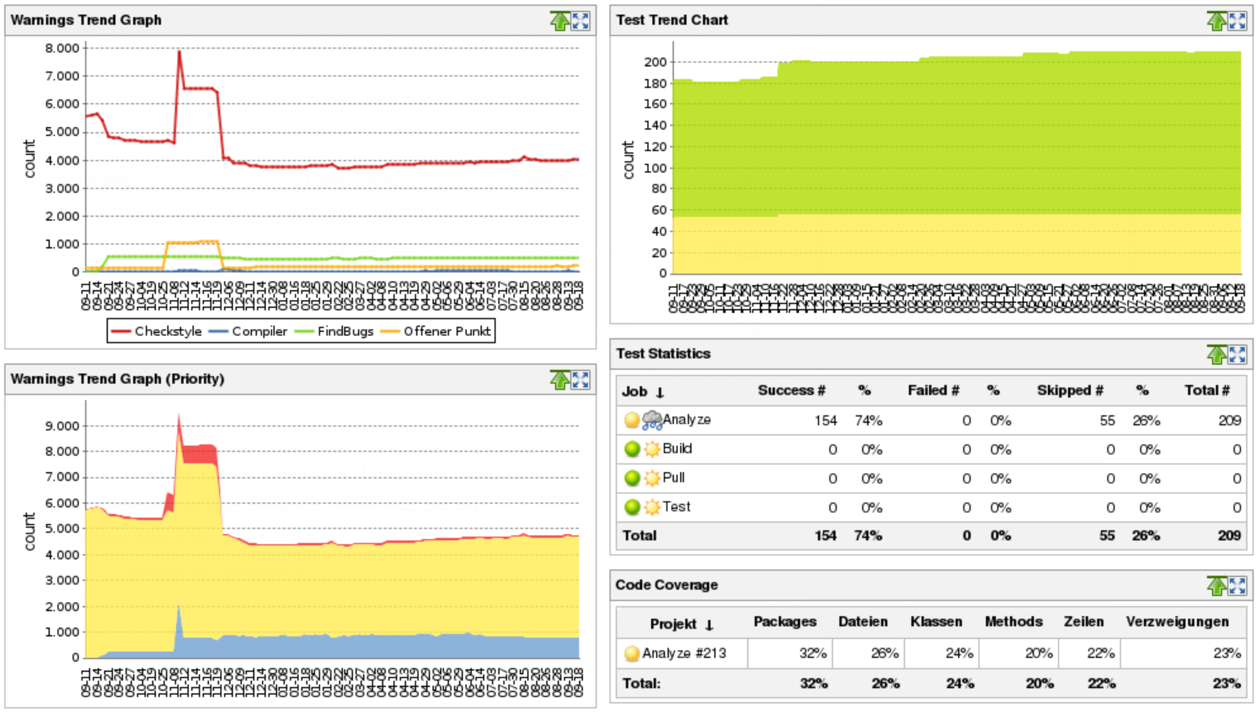
Symbol: S M L

Legende RSS Alle Builds RSS Nur Fehlschläge RSS Nur jeweils letzter Build

Figure 4 – Jenkins Build Pipeline

The very first widget on the Dashboard is the Build Pipeline as shown in Figure 4. Each of the elements in this widget, except the first orange item which identifies the pipeline instance, represents a different build/ test phase called job. Green items have been completed successfully in the last run, yellow items have not be run, and red items failed to complete. The text on the elements reference detailed views about the phases' subitems. The referenced views also allow to navigate to a configuration view where the jobs execution can be changed if the user has the appropriate access rights.

Below the schematic of the pipeline lies a job summary. Each row in the list represents one of the jobs in the pipeline. In the first two columns the status of the last is opposed to the current build with easy to recognize pictograms. The status indicators are followed by some timing values and are concluded by a button to trigger the job and one to display its console output.



Test Statistics

Job	Success #	%	Failed #	%	Skipped #	%	Total #
Analyze	154	74%	0	0%	55	26%	209
Build	0	0%	0	0%	0	0%	0
Pull	0	0%	0	0%	0	0%	0
Test	0	0%	0	0%	0	0%	0
Total	**154**	**74%**	**0**	**0%**	**55**	**26%**	**209**

Code Coverage

Projekt	Packages	Dateien	Klassen	Methods	Zeilen	Verzweigungen
Analyze #213	32%	26%	24%	20%	22%	23%
Total:	**32%**	**26%**	**24%**	**20%**	**22%**	**23%**

Figure 5 - Build Statistics

While the indicator based status of the build and test run is fine to notice build failures and see the overall status at a glance, it is not sufficient to measure the software quality of the system. The statistics widget as shown in Figure 5 shows the trend of various metrics over time. The type of measures available in the charts can be divided into static and dynamic analysis of the software. The static metrics are aggregated in the two left charts, while the dynamic metrics are located on the right. In the current analysis job the tools Checkstyle[1] and FindBugs[2] determine violations of the code style guidelines and statically determinable faults and bad practices. Additionally compiler messages and TODO entries are evaluated. The dynamic or runtime metrics depend solely on tests. This includes unit test, as well as integration and system tests which are aimed in this document. Besides the obvious numbers of passed and failed tests, a code coverage graph which is backed up by the tool Cobertura[3] is available. In the future, this also may be the place for the requirements traceability based on the OASIS Test Assertions described in Section 2.1.

[1] http://checkstyle.sourceforge.net/
[2] http://findbugs.sourceforge.net/
[3] http://cobertura.github.io/cobertura/

Document name:	SP4/ WP45					**Page:**	13 of 27
Reference:	https://dms-prext.fraunhofer.de/livelink/livelink.exe/overview/3329324	**Dissemination:**	PU	**Version:**	1.1	**Status:**	Final

3.1.2 Jobs of the Pipeline

The jobs in the Build Pipeline fulfill different purposes. The first group of jobs is responsible for the build of the software components, and local tests and analysis belonging to unit and integration tests. This group can be performed frequently e.g. with each change in the source code. The second group connects the different components and monitors whether they are functioning properly.

3.1.2.1 Pull

At the moment the first step of the build pipeline is the pull step. In this step, the code gets pulled from the central code repository. After pulling the whole build pipeline gets triggered. This step gets repeated every time something has changed in the code repository.

3.1.2.2 Build

After the pull step, the code gets built in this step. The following JDKs are used to build the components:

- OpenJDK 6
- OpenJDK 7
- OracleJDK 7 u7 x64
- Oracle JDK 6 u35 x64

These different JDKs can also be extended during the lifetime of the project.

3.1.2.3 Test

For the unit tests the tool TestNG is used. With this plug-in the different unit tests of the components run on different JDKs. This ensures that the code not only runs without any problems on the JDKs, the developers have installed on their machines.

Unit tests and Integration tests run during this step. Even if there are errors during this step, the next step called "Analyze" will run.

3.1.2.4 Analyze

For the analyzing of the code which is located on the repository different Jenkins plugins will be used. An overview of warnings and errors are aggregated within a graph which shows the history during the lifetime of the project. Figure 6 shows the graph which gives a quick overview on the number of checkstyle errors, compiler errors and other important aspects in terms of software quality.

Compiler warnings

During the build process maven generates several compiler warnings, which get collected and are included in this category of errors.

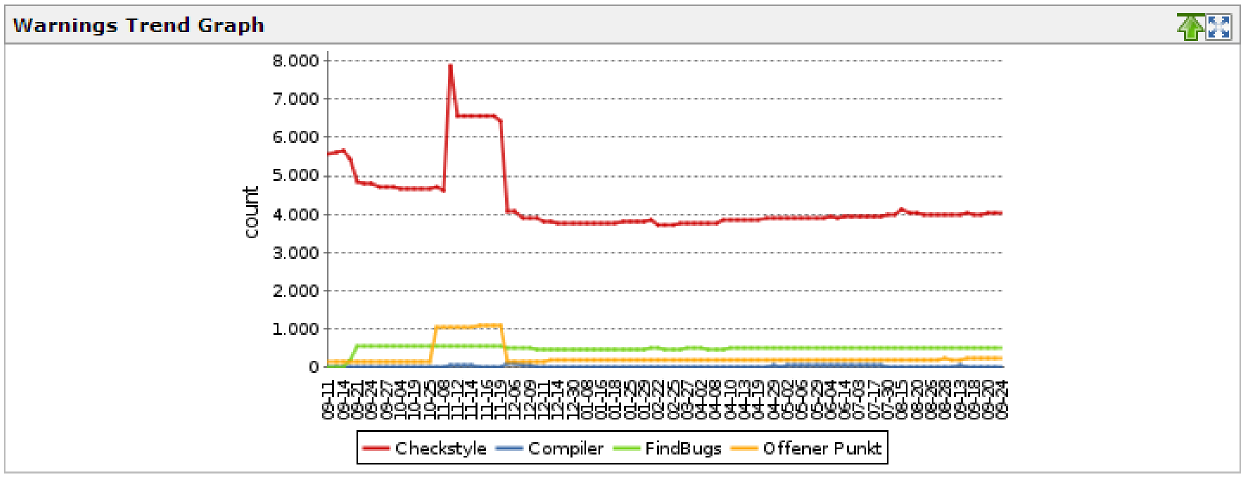


Figure 6 - Test Trend

Unresolved tasks

The developers can use different tags to indicate unresolved tasks within the source code. These tags get parsed with the plugin "Task Scanner". The different tags are still to be discussed and will be promoted to the developers of the project.

Test coverage

The plugin cobertura is used to show the test coverage of the several components which are developed. There is an easy to read overview of the different classes and their respective test coverage.

3.1.2.5 Deploy

Currently the deployment of the server artifacts is done manually. In future there may be ways to automate the deployment of the software artifacts on a test environment automatically.

3.1.2.6 System Test

For the system tests the tool Sikuli is used. For this reason we set up a second Jenkins which runs on a Windows 8 machine and runs these tests on a timely basis with smart cards attached to the server. As Sikuli has the need to have a real graphical user interface it is not possible at the moment to also include this type of tests within the main Jenkins instance.

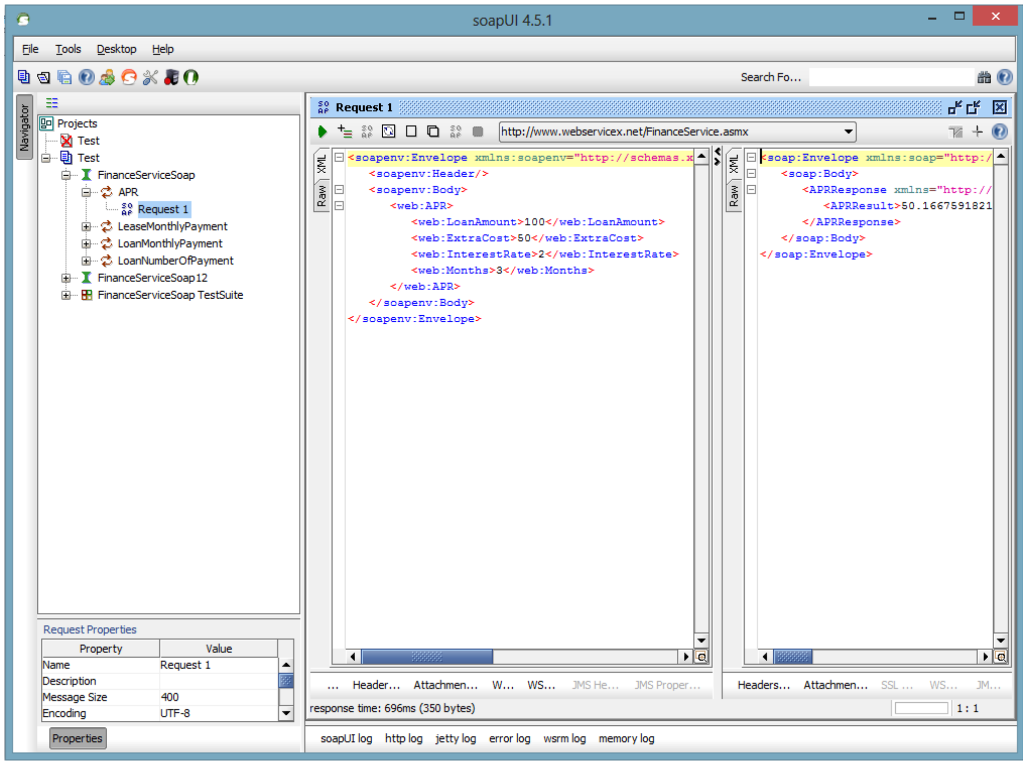
3.1.2.7 Requirements Trace

During the lifetime of the project the linking of the requirements which will be documented in the central wiki platform may be linked to specific test results on the Jenkins platform. This will improve the traceability of requirements and the results from the system tests drastically. It is intended to provide a view on requirements which are not covered by tests and also provide a view on requirements which are covered but are not fulfilled at the moment.

The outcome of this approach is an easy overview on topics which need to be covered by new tests.

3.2 SoapUI

SoapUI is an open source tool for functional testing of web services. It supports REST and SOAP interfaces and will be used during the project to test the software components functions.


soapUI 4.5.1
File Tools Desktop Help
Search Fo...
Navigator
Projects
Test
Test
FinanceServiceSoap
APR
Request 1
LeaseMonthlyPayment
LoanMonthlyPayment
LoanNumberOfPayment
FinanceServiceSoap12
FinanceServiceSoap TestSuite
Request 1
http://www.webservicex.net/FinanceService.asmx
XML
Raw
Header... Attachmen... W... WS... JMS He... JMS Proper...
Headers... Attachmen... SSL ... WS... JM...
1 : 1
soapUI log http log jetty log error log wsrm log memory log
Properties

Request Properties

Property	Value
Name	Request 1
Description	
Message Size	400
Encoding	UTF-8

```
<soapenv:Envelope xmlns:soapenv="http://schemas.x
   <soapenv:Header/>
   <soapenv:Body>
      <web:APR>
         <web:LoanAmount>100</web:LoanAmount>
         <web:ExtraCost>50</web:ExtraCost>
         <web:InterestRate>2</web:InterestRate>
         <web:Months>3</web:Months>
      </web:APR>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap:Envelope xmlns:soap="http:/
   <soap:Body>
      <APRResponse xmlns="http://
         <APRResult>50.1667591821
      </APRResponse>
   </soap:Body>
</soap:Envelope>
```

response time: 696ms (350 bytes)

Figure 7 - SoapUI main interface

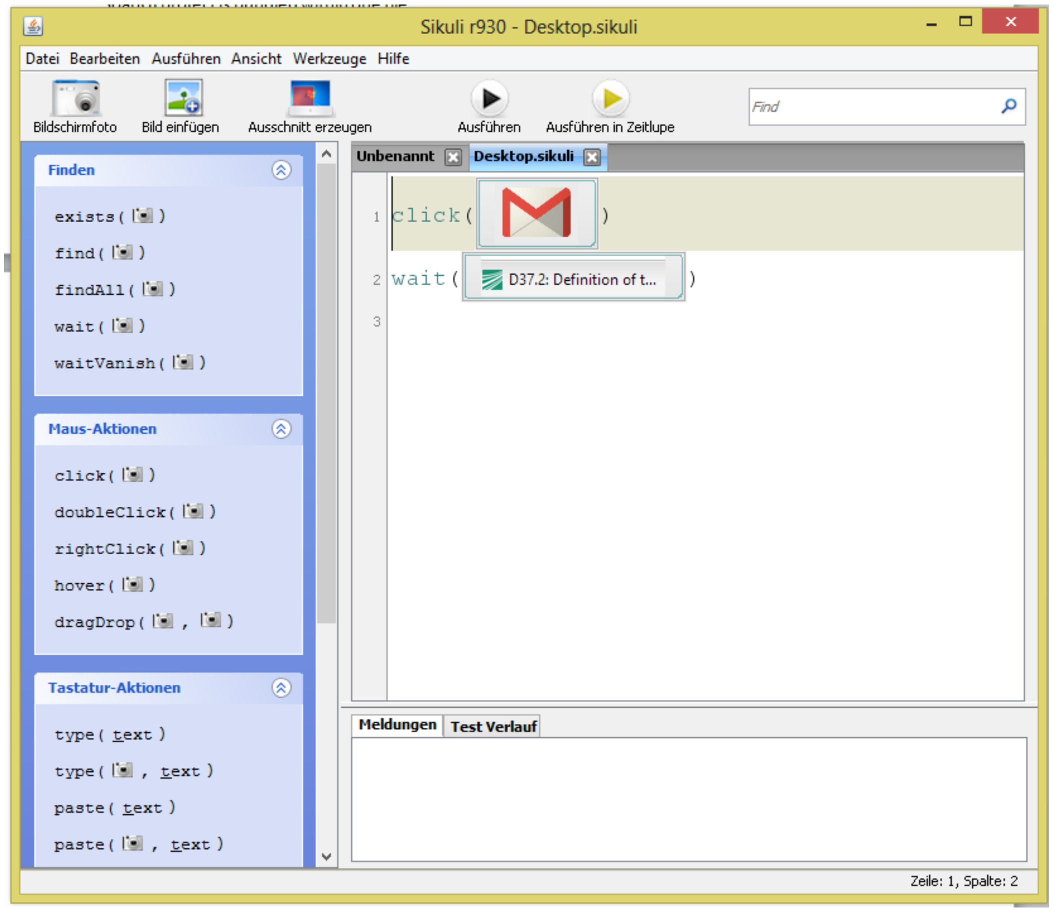
Figure 3 shows the main interface of SoapUI. In order to have a maintainable suite of tests for the different server components, a svn repository will be used. On this repository different SoapUI projects will be saved and versioned.

To avoid the need to buy SoapUI pro version, the different software components will have responsible test managers, who must coordinate the testing activities for each component. Each SoapUI project is bundled within one file.

3.3 Automated system tests

To test the communication throughout the whole *FutureID* infrastructure, system tests will be defined and run on a timely basis. These system tests will use real service providers and their integration of *FutureID*. As the work package WP37 is concerned with tests for the client platforms, these tests will cover mainly topics of the server infrastructure. Therefore different states of the server infrastructure will be mainly tested by the test cases.

For PC platforms the tool Sikuli [2] is an option for real system tests driven by image recognition and simulated interaction. In [?] authors evaluated the tool Sikuli and compared it to a commercially available tool. Overall the Sikuli was rated well in comparison to the commercially available tool for 10.000€ license fee. The authors were able to save up to 78% of execution time for the automated tests in comparison to manual tests.



```
exists( )
find( )
findAll( )
wait( )
waitVanish( )
click( )
doubleClick( )
rightClick( )
hover( )
dragDrop( , )
type( text )
type( , text )
paste( text )
paste( , text )
```

```
click( )
wait( D37.2: Definition of t... )
```

Figure 8 - Sikuli scripting

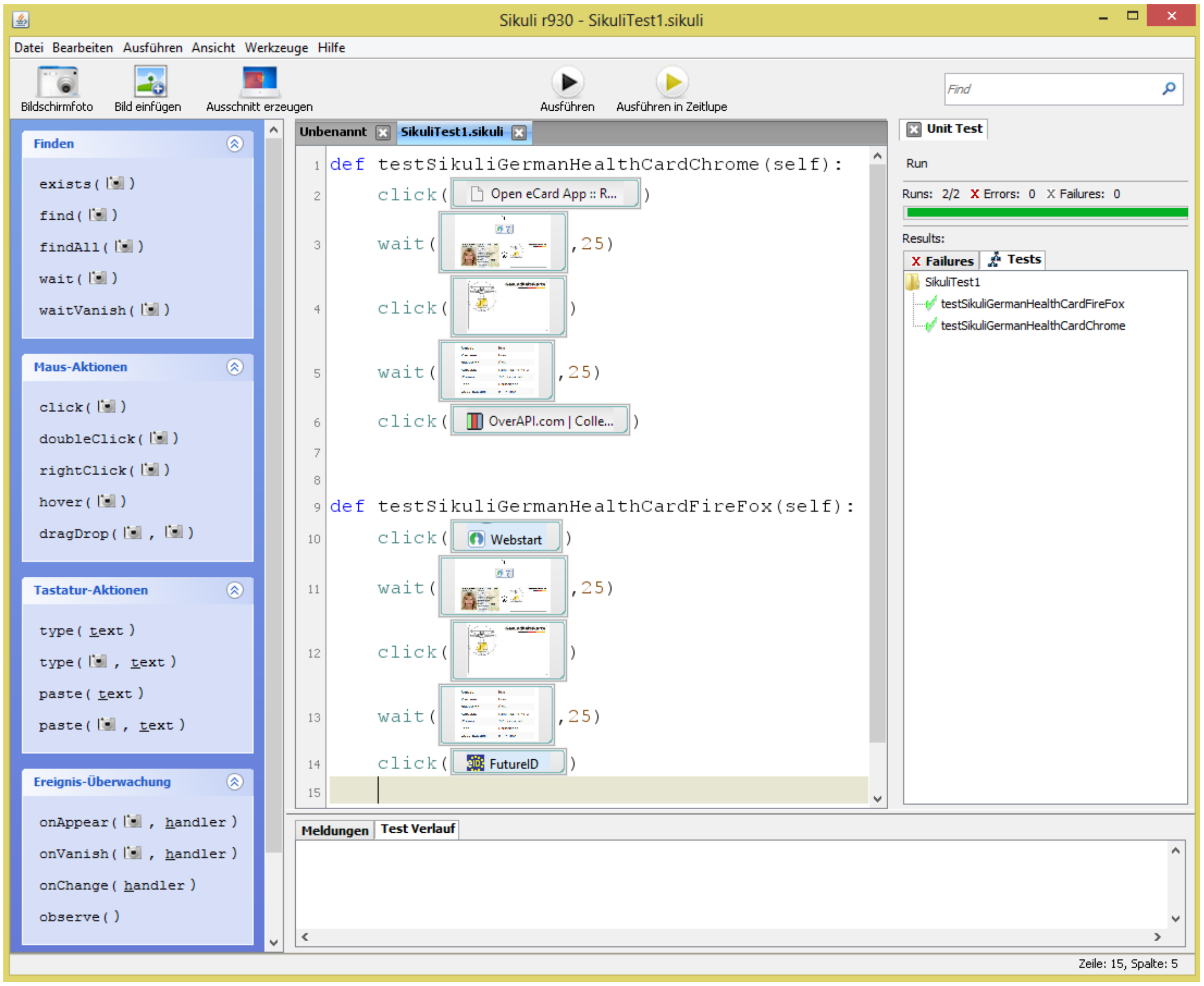
Figure 4 shows the start screen of Sikuli. Testers can use this tool to automatically perform tasks which a daily user would perform by using the *FutureID* infrastructure. This leads to real and authentic interaction with the overall system.

By drag and drop actions can be defined, which later on build the script which can be performed. The actions are divided into three separate groups. The first group stands for all actions which can be used to search for certain elements on the screen. The script could for example wait until the *FutureID* client gets opened and showed on the screen. The second group comprises all mouse actions. Elements can be clicked, double clicked, right clicked, hovered and dragged and dropped. The third group is used for any keyboard actions. This kind of interactions can also be used to start the windows search or for opening the specified browser before browsing to the target service.

To evaluate the solution with Sikuli we created a test case which runs a demo authentication with the german healthcard with different browsers. This test script is shown in Figure 5. With just very short test scripts it was possible to start the service and evaluate the presented results.

The upper part of the test script defines a test for the Chrome browser. The Sikuli engine first clicks on the respective bookmarks, waits until the desired image shows up and finally it clicks on the card symbol. Afterwards the presented result gets analyzed. In case one of the tests fails, the whole test will be marked as “FAILED”. Otherwise the test will be marked as “PASSED”.

This way, all defined test cases can be automated and run automatically on a daily or weekly basis. As Sikuli needs to be used on operating systems with a graphical user interface it is not possible to run these tests on the same machine on which the main Jenkins instance runs.



```
exists( )
find( )
findAll( )
wait( )
waitVanish( )

click( )
doubleClick( )
rightClick( )
hover( )
dragDrop( , )

type( text )
type( , text )
paste( text )
paste( , text )

onAppear( , handler )
onVanish( , handler )
onChange( handler )
observe( )
```

```
def testSikuliGermanHealthCardChrome(self):
    click( Open eCard App :: R... )
    wait( ,25)
    click( )
    wait( ,25)
    click( OverAPI.com | Colle... )


def testSikuliGermanHealthCardFireFox(self):
    click( Webstart )
    wait( ,25)
    click( )
    wait( ,25)
    click( FutureID )
```

```
Runs: 2/2  X Errors: 0  X Failures: 0
Results:
X Failures  Tests
SikuliTest1
  testSikuliGermanHealthCardFireFox
  testSikuliGermanHealthCardChrome
```

Figure 9 - Sikuli Testcase

4. Test scope

The test scope is one very important aspect of the test environment. On one hand the test scope should be as wide as possible to test as many aspects and configuration of the server as possible. On the other hand the test scope needs to be maintainable on the long run. A scope which is set too broad will lead to sporadic testing behaviour during the project.

4.1 *FutureID* components

In the following the different *FutureID* components will be described. It is important to understand the architecture of the server components to be able to define relevant test vectors and the underlying infrastructure.

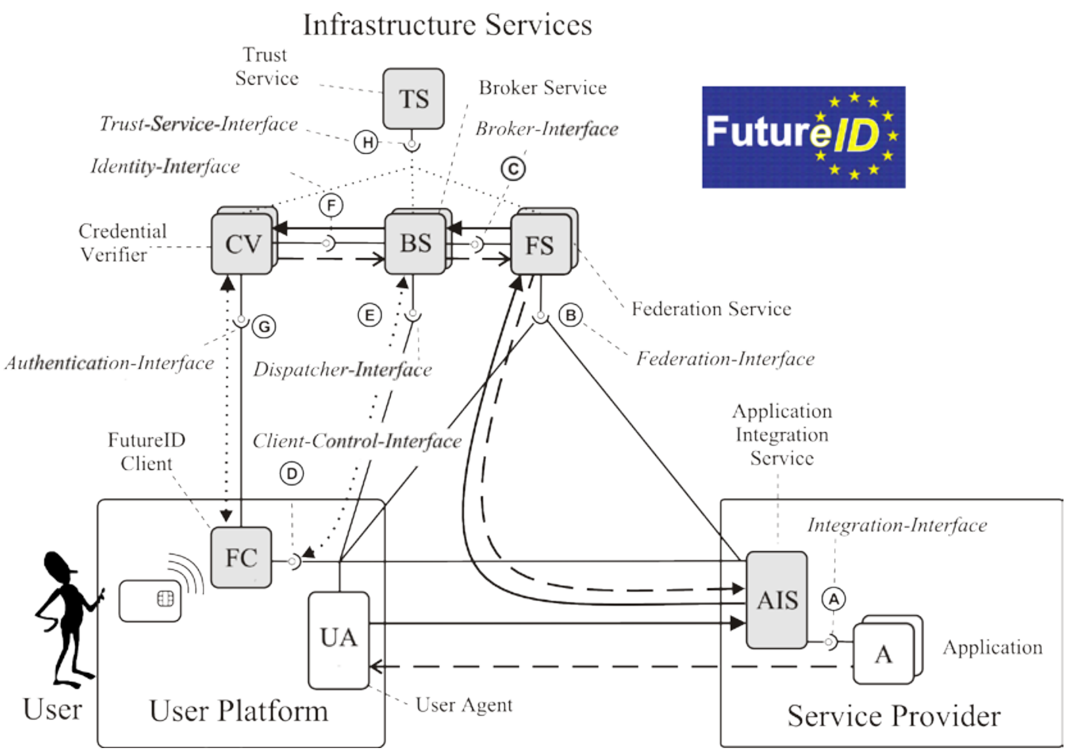


Figure 10 - Architecture

Identity Providers

The identity providers create, maintain and manage the different identities of the different users. The Identity Providers also provide the user authentication to a service provider. The identity providers are usually not under the influence of the *FutureID* project team, however will be covered within different automated system tests. Different identity providers also use different authentication methods.

Service Providers

The Service Providers need the information provided by the respective Identity Providers to provide its services to the users. This will usually refer to a website or other application provider facilities. Often times, different services are bound to certain users. In order to present the relevant information only to the right users, the user information of the identity provider is used by the service providers.

Broker Service

A major advantage of the usage of *FutureID* is that the service providers only have to implement *FutureID* to be able to use many different Identity Providers. To be able to deliver this main advantage, the Broker service is needed. Its task is to make it possible to use different authentication tokens and to find the right Identity Provider for the selected authentication method. It is the intermediary between the Universal Authentication Service (UAS) and the Federation Service.

There will be two different modes of the Broker service. With the dispatcher mode the Broker service will only dispatch and determine an appropriate credential verifier. The more sophisticated and more complex mode is called claims transformer mode. In this mode, the Broker service performs the authentication itself and transforms the claims to the format demanded by the service provider.

Universal Authentication Service

The Universal Authentication, will be able to support various authentication protocols implemented by different authentication tokens. The goal is to be able to support all authentication protocols implemented by the various authentication tokens that are deployed across Europe. This includes the existing eID cards, eHealth cards and electronic signature cards.

Application Integration Services

The Applications are integrated into the *FutureID* infrastructure via the Application Integration Services (AIS) and consumed by the User.

The presence of mechanisms, protocol versions and compliance will be assessed for each major version.

4.2 Supporting infrastructure

In order to thoroughly test the *FutureID* server components, services and functionality, a proper supporting test infrastructure needs to be in place. This includes components of PKI (Public Key Infrastructure) components or at least partly functional components simulating the same functionality. Following key components or similar functionality are part of a typical PKI infrastructure and should be in place:

- Validation Authority - OSCP [3] or (CRL) Certificate Revocation Lists
- Certificate Authority or another source for certificates
- Directory Service - LDAP
- Validation Authority
- Initial Trust Vector
- Registration Authority
- Access credentials
- ID tokens
- Working client installations
- Error provocation tools that can intentionally disturb network protocols (e.g. a packet sniffer deleting/changing requests to and from a validation service)

4.3 Test vectors

Test vectors describe a set of inputs for test cases. A comprehensive set of these vectors have to be defined to allow for efficient and thorough testing. Test vectors for testing the *FutureID* server infrastructure need to include corrupted as well as valid data to be used as input into the various systems and protocols. For testing certificates, following test vectors should be considered:

Credentials:

- Different certificates standards (x509, etc.)
- Broken and corrupted certificates, data and files (e.g. card info files)
- Expired or revoked certificates
- Broken subject → issuer relationship (e.g. the issuer didn't issue this certificate)
- Broken subject → issuer relationship somewhere in the chain (walking through the subject → issuer chain

Formats:

- Security Assertion Markup Language (SAML) [6]: SAML was created by the OASIS Security Services Technical Committee and is a standard for the exchange of authentication information based on XML. In FutureID, it is planned to use this protocol for communication between identity providers and the service providers. It is often used for single sign-on functionality in the browser, distributed transactions and credential verifiers using third party identity providers.
- OpenID: OpenID is an open standard for authenticating users by a third party identity provider service that can be adopted by websites in order to "outsource" authentication and digital identity functionality. OpenID [7] isn't restricted to using a specific of credential but allows using different credentials such as passwords, PKI-Cards or biometrics.

Protocols:

On a lower level, the protocols used for communication and authentication between components need to be tested and test vectors need to be defined with valid and corrupted input data. Also networking problems such as latency, missing packets should be considered. For security testing, checksums should be validated and tested to identify manipulated data. Following protocols may be relevant and should be included in the testing efforts:

- Online Certificate Status Protocol (OCSP): OSCP is an Internet protocol described in RFC 6960 for checking and obtaining the revocation status of an X.509 digital certificate in a request/response mechanism much like http.
- Transport Layer Security (TLS): [8] is the successor of the Secure Sockets Layer (SSL). It is a protocol to provide secure communication over the Internet using cryptography. Asymmetric cryptography is used for authentication of key exchange, while symmetric encryption is used to provide confidentiality and message integrity.

Policy languages:

To ensure correct transformation of claims, and valid use of identity tokens against service providers, policy assumptions must be made. Such policies will be used in trust evaluation, and under transactions within the Broker service.

- Presence of policy languages to express CA policies, privacy policies, etc.
- Mechanisms for policy conformance checking

5. Conclusion

In this deliverable the outline of a reference test environment for the *FutureID* server infrastructure was presented. The required test infrastructure consisting of an adequate toolset for testing the functionality and communication between the *FutureID* components was identified and their purpose in the server test environment was illustrated. Test vectors, covering various input credentials, protocols and formats were listed and specified. Test cases involving corrupted and invalid input data or credentials should be considered in order to ensure a reliable and usable platform in real life scenarios.

6. References

[1] “OASIS -Advancing open standards for the information society” [Online]. Available: https://www.oasis-open.org/. http://www.sikuli.org/

[2] “Jenkins - an extendable open source continuous integration server„ [Online]. Available: http://jenkins-ci.org/

[3] “Sikuli Script" [Online]. Available: http://www.sikuli.org/.

[4] “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP„ [Online]. Available: http://www.ietf.org/rfc/rfc2560.txt

[5] “SoapUI„ [Online]. Available: http://www.soapui.org/

[6] “OASIS Security Services (SAML) SAML„ [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security http://openid.net/]

[7] “OpenID „ [Online]. Available: http://openid.net/]

[8] “The Transport Layer Security (TLS) Protocol Version 1.2„ [Online]. Available: http://tools.ietf.org/html/rfc5246]