WP37 – Client Testbed

D37.5 – Integration Testing Tools

Document Identification	
Date	26.10.2015
Status	Final
Version	1.0

Related SP / WP	SP3 / WP37	**Document Reference**	D37.5
Related Deliverable(s)	D37.1, D37.2, D37.3, D37.4, D37.6	**Dissemination Level**	PU
Lead Participant	SK	**Lead Author**	Mart Toom
Contributors	Janina Hofer (USTUTT) Trenton Schulz (NRS)	**Reviewers**	ULD, AG

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

Abstract

Different FutureID components must not only function by and for themselves and for their purpose only but must also interoperate with other modules to fulfil the overall functionality of the FutureID infrastructure. The FutureID infrastructure comprises 57 modules that build the entire client code base. Not all of them require integration tests, for example because their only purpose is to combine other modules together; so a priori they do not need integration tests. However, other modules or add-ons, such as e-signing plugins, PIN management, or PKCS11, need to properly interact with e.g. the GUI or the add-on framework. The integration testing environment does not differ drastically from the module testing environment. The same tools, such as Jenkins, Maven, Git, TestNG and Mockito, are provided by the testbed. It is absolutely important to assign the different tests to the correct group to define how they should be executed. The integration tests therefore must be annotated with “integration”. Keep in mind that complex integration tests are usually labeled with more than one annotation, since different elements and modules are integrated in one test. The focus of this present document is on the assigned annotation and test execution approach within FutureID as well as on the test result presentation.

Document Information

Contributors

Name	Partner
Mart Toom	SK
Trenton Schulz	NRS
Janina Hofer	USTUTT

History

Version	Date	Author	Changes
0.1	20.08.2015	Mart Toom	Created document structure
0.2	18.10.2015	Mart Toom	Added Integration testing tools
0.3	18.10.2015	Janina Hofer	Introduction, Abstract, Summary
0.4	22.10.2015	Mart Toom	Including comments after internal review, last corrections
1.0	26.10.2015	Mart Toom	Included feedback from reviewers.

Table of Contents

1. Introduction

1.1 Project Scope

The FutureID project builds a comprehensive, flexible, privacy-aware and ubiquitously usable identity management infrastructure for Europe, which integrates existing eID technology and trust infrastructures, emerging federated identity management services and modern credential technologies to provide a user-centric system for the trustworthy and accountable management of identity claims.

The FutureID infrastructure will provide great benefits to all stakeholders involved in the eID value chain. Users will benefit from the availability of a ubiquitously usable open source eID client that is capable of running on arbitrary desktop PCs, tablets and modern smart phones. FutureID will allow application and service providers to easily integrate their existing services with the FutureID infrastructure, providing them with the benefits from the strong security offered by eIDs without requiring them to make substantial investments.

This will enable service providers to offer this technology to users as an alternative to username/password based systems, providing them with a choice for a more trustworthy, usable and innovative technology. For existing and emerging trust service providers and card issuers FutureID will provide an integrative framework, which eases using their authentication and signature related products across Europe and beyond.

To demonstrate the applicability of the developed technologies and the feasibility of the overall approach FutureID will develop two pilot applications and is open for additional application services who want to use the innovative FutureID technology

Future ID is a three-year duration project funded by the European Commission Seventh Framework Program (FP7/2007-2013) under grant agreement no. 318424

1.2 Deliverable Scope and Outline

This present document is concerned with integration testing of the FutureID infrastructure. It presents an interoperability test framework for implemented modules from other implementation tasks.

Firstly, an overview about integration tests in general and in the context of FutureID, including components to be tested and test criteria, is given in Chapter 2. The subsequent Chapter 3 elaborates on the developed integration testing infrastructure. The overall testing environment, including testing and result presentation tools such as TestNG, is explained in detail in this section.

The results of the extensive integration testing are presented and explained in the context of Chapter 5. This deliverable is completed and summarized in Chapter 6 in which also a short outlook is given.

1.3 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

2. Integration Tests

The FutureID client codebase comprises over 50 different modules. Developers work independently on these modules, and the modules must eventually work together. So, there needs to be some way to test that these modules can work together. Integration tests are the projects solution to this task.

2.1 Integration Test Overview and Purpose

As defined in D37.2 [2], testing is divided into three different groups. There is unit testing where functions are tested in a single module. Integration testing examines the functionality between two or more modules. There is also system testing, which builds all the components and runs it on systems that the client should run on. Finally, there is acceptance testing. As described in D37.6 [3], we chose to focus on the ready for release, installation, and robustness criteria instead of a standard user test as that was covered by multiple tasks in the project, such as the Usability evaluation.

2.1.1 Components to be Tested

There are 57 different modules that build the FutureID client codebase:

- FutureID Client
- Open eCard Webservice Definitions
- class-list Plugin
- Open eCard WS common
- Open eCard WS classes
- Open eCard WS classes FutureID
- JAXB Marshaller
- Open eCard I18n
- Crypto packages
- Open eCard Bouncy Castle
- Open eCard Common Libs
- GUI implementations
- GUI common
- CardInfo files
- IFD implementations
- IFD common
- SmartcardIO implementations
- PC/SC for OS X
- PCSC SmartcardIO
- Swing GUI
- IFD core
- Card Recognition
- Add-on Framework
- Crypto common
- SAL
- SAL common
- Android Marshaller
- Open eCard I18n FutureID
- Open eCard TLS
- Transport
- Dispatcher
- Apache shaded HTTP core
- Open eCard HTTP core
- Event Manager
- Tiny SAL
- Management
- Integrated Add-ons
- TR-03112 Add-on
- Status
- Control Interface Bindings
- HTTP Binding
- HTTP Binding FutureID
- Graphics
- About Dialog

- Graphics FutureID
- About Dialog FutureID
- RPC GUI
- IFD protocols
- PACE protocol
- NFC SmartcardIO
- PIN Management Add-on
- Generic cryptography protocol
- Client Implementations
- Open eCard Java Version Checker
- Rich Client
- Rich Client FutureID
- Rich Client FutureID RPC

Some of these modules are simply a way of combining other modules together, so a priori they do not need integration tests. For example, the About Dialog modules are simply called from within the system tray application in the same way they would have been called if they were standalone. Some modules are more specific implementations of a base. Several modules add FutureID branding and are based on another module. These modules do not need an integration test with their base module, but they need testing in place where they are used. Yet, modules like the different protocols should have an integration test with the UI infrastructure to make sure that they run correctly. The remote procedure call (RPC) modules need integration checking to make sure that they work with the GUI and the other modules.

There are several different add-ons that need to be checked against the add-on framework and in the GUI. This includes the e-signing plugins, PIN management, and PKCS11.

2.1.2 Testing

All integration tests have been written using TestNG, the same testing framework that was used for writing unit tests. TestNG qualifies for integration tests because most of the interfaces would work similar to unit tests. Using TestNG offers the additional advantage that developers were already familiar with the testing framework when creating the integration test.

3. Integration Testing Tools

As pointed out in D37.2 - Test Strategy [2] document, writing tests on code level – unit tests and component integration tests is mainly the task for component developers. Like using unit tests is quite powerful tool for developers to gain confidence about the quality of written or reworked code, integration tests provide similar confidence beyond the borders of single modules or units.

In the following chapters we describe integration testing tools provided to component developers by FutureID client testbed. As the tools used for integration testing by developers are quite the same as the tools for unit testing, we provide in this document a lighter overview of the tools and setup to get started with integration testing. Additionally we bring out the additional complexities that Integration tests have. In case a more deeper view to unit test environment setup is needed, one should to take a look in deliverable D37.4 - Module TestTools [3].

3.1 Integration Testing Environment

Integration tests serve a software project on many different levels. First, they are useful and needed tools when developers are creating the software and many smaller components are tied to bigger piece of working code. In this case, Integration tests are executed in development environment by developers and are providing direct value and feedback to developers. This phase of project requires that chosen tools can be easily integrated to development environment.

Secondly, Integration tests are providing value once different pieces of code are put together and the system should be able to work smoothly as a whole. In this case, integration tests should be able to work in integration environment that in case of FutureID project is based on Jenkins continuous integration server. In the FutureID project, executing tests and providing information to interested parties is the task of testbed. In the following chapters there will be given overview of different components that together compose the Integration testing environment of FutureID client testbed.

3.1.1 Jenkins

Jenkins (http://jenkins-ci.org) is a continuous integration platform with a modular architecture and the ability to extend the functionality via plugins. It is the main entry point for testing and evaluating test results. Therefore all further tools will be integrated in Jenkins. Furthermore, all test results will be collected and presented on this platform. Jenkins provides a summary of the important information on the main view for all participants. [2]

As integration tests have often more preconditions to be able to run successfully, those tests cannot be added directly to test pipeline of FutureID testbed. For integration tests there are defined dedicated test jobs in Jenkins of FutureID testbed. Those Integration test jobs can be executed manually by user once it is known that all required preconditions are fulfilled.

3.1.2 Maven and the Project Object Model (POM)

In large scale projects like FutureID client consisting tens of subprojects and components, it is essential to use such a central building and dependency management tool that can handle fluently complexity of the system. For FutureID project Maven is chosen to serve as such a integration and building tool.

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information. [5]

Jenkins runs on top of established Java technologies like Maven. Maven is a tool for software management and comprehension. Using the project object model (POM), an XML file that describes the project and actions that can be performed, Jenkins can execute all included test cases and assertions. Maven can do different tasks (or goals in its own language), such as building a project, downloading dependencies, generating documentation, or executing scripts. Jenkins can call maven directly and execute goals that are in a POM file. [4]

3.1.3 Git Repository

Many distributed teams contribute to the FutureID infrastructure. This is also the case for the test development, which e.g. includes already implemented test cases. Eventually, it is necessary to follow a structured approach to save and share files and data among the team in order to avoid unnecessary double work or in a worst case the loss of important data. Therefore, a Git repository is used as the central tool to avoid the above mentioned problems. [4]

Git is a distributed source control management (SCM) written for performance and easy branching in mind. It is an open source system that was created after the Linux project lost the ability to use the BitKeeper SCM. The founder of Linux, Linus Torvalds, wanted to have the same capabilities that he had with BitKeeper, but there was no comparable open source variant (at least performance-wise). So, he wrote the initial version of Git. Other systems, like Mercurial, appeared soon after Git that have very similar performance and features, but Git is the most popular system. [4]

As Git is in use as general distributed code repository for the FutureID project and Integration tests live closely togethere with application code, it is the only possible logical way to use Git for Integration tests.

3.1.4 TestNG

TestNG [5] is a powerful testing framework designed to simplify a wide range of testing tasks.Therefore it supports several test levels like unit testing and integration testing. It was mainly inspired by JUnit and NUnit but introduces several new functions like support for data-driven testing, support of test parameters, usage of dependencies between test methods as well as additional powerful annotations. TestNGs very flexible configuration mechanism allows to specify arbitrary methods to be invoked at particular moments during your test run, for example "before every test method" or "after all the test methods have run".

As there are several IDE-specific TestNG-plugins available, it is easy to use TestNG with the most common development environment. The developer can choose or continue using his favoured working environment. For example a plugin for Eclipse can be found at [6].

For using TestNG in FutureID client testbed environment, special plugin is integrated into Jenkins environment for presenting TestNG Unit test results. Same mechanism can be utilized also for presenting Integration test results.

3.1.5 Mockito

One problem when writing Integration tests is the availability of other components or interfaces. Although one can call or execute multiple components to check the correctness of integration between them, there is often something missing. In such a cases there can be used fake or mocked interfaces or components that replace the missing part of the system. There has been created a tool Mockito [7] to simplify the creation of fake interfaces and classes required in Java code testing.

Mockito is an open source mocking framework extending automatic testing in Java. It enables to simulate classes together with their functionality (mock objects) without implementing their entire functions and variables. For example, the developer can decide on its own, which values should be returned by certain method calls. It is also possibly to make a partial mocking of objects (spy objects) where the original methods are invoked if they are not mocked. Thus, Mockito allows easy creation of independent and very focused tests for classes and methods, which normally can not be isolated within Unit or Integration tests. With Mockito, it is also possible to create interface mock ups to simulate handling of external resources or classes. [8]

4. Creating Integration Tests

In this chapter, we give an overview of how to create such tests that are executed as integration tests in FutureID client testbed.

The aim is not to teach how to write Unit or Integration tests, but rather how to mark a test as Integration test.

More detailed overview of how to annotate and what possibilities are provided by Unit testing tools of FutureID client testbed can be acquired from [3].

4.1 Defining TestNG Test Suite for a Component

By defining the TestNG test suite for a component, one can specify which test groups are available for annotating the tests and how the groups are related to each other.

See sample test suite definition file, that defines test groups for a component.

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="PKCS11 Module TestSuite" parallel="false">
 <test name="PKCS11 Addon Tests">
     <groups>
        <define name="module">
            <include name="checkin"/>
        </define>
        <define name="all">
            <include name="module"/>
            <include name="integration"/>
            <include name="gui"/>
        </define>
        <define name="non-gui">
            <include name="module"/>
            <include name="integration"/>
        </define>
        <run>
            <include name="module"/>
        </run>
    </groups>
    <packages>
        <package name="org.futureid.client.addons.pkcs11" />
    </packages>
  </test>
</suite>
```

Figure 1 - Test suite definition of PKCS11 module.

It is reasonable to stick to same set of groups throughout one project to provide consistency. For FutureID project following groups are available:

- checkin – this group should be used for all tests that should be executed before code is committed to central Git repository
- module – module tests contain in addition to checkin group also additional tests that could be executed in nightly tests.
- gui – this group specifies tests that require graphical user interface for execution
- integration – this group specifies tests that are testing integration of different modules
- Non-gui – this group contains all tests that doesn't require graphical user interface for execution
- all – contains all tests

4.2 Annotate the Tests

Once the developer has finished writing the test, it should be decided which groups should be assigned to tests.

Any test that is running against more than one single module or unit in program code is by definition already potentially integration test. But it should be developer's wise decision if the test is really meant to be integration tests or is it just extended unit test.

Once the decision is made, then for integration tests group label "integration" should be used.

Below is one example of annotating test as integration test

```
public class SampleAnnotationTest
{
   @Test(groups = { "integration" })
   public void integrationTestMethodOne() {
      System.out.println("Integration test one");
   }
   @Test(groups = { "module", "gui" })
   public void testMethodTwo() {
      System.out.println("Test with 2 groups: module and gui");
   }
}
```

While adding the annotations to tests, one should keep in mind that one test can have one or many groups assigned. As more complex tests like integration tests usually are, can for example contain also running GUI elements and therefore tests should be labelled to belong to both - "integration" and "gui" groups.

4.3 Executing Tests

4.3.1 Executing Integration Tests from Command Line

Once tests are written and correctly annotated, there is the possibility to execute tests by groups. For example, before committing the code a developer might want to run only tests that are running fast and required to be executed before commit (tests in group “checkin”). The group to be executed can be specified either in the Maven POM file or by giving the command line option to Maven command. The latter is a preferred option as it provides more flexibility. For running integration tests from command line, the following command could be used in the project folder:

mvn test –Dgroups=integration

Also, multiple groups can be specified by providing comma-separated list

mvn test –Dgroups=checkin,integration

4.3.2 Executing Integration Tests in Jenkins

For executing integration tests in the FutureID testbed, there is a defined dedicated test job for executing integration tests related to the FutureID client. As the Jenkins system is also internally using Maven for executing tests, the main principle for specifying a test group is the same. Maven should be told to use integration tests by specifying group parameter *–Dgroups=integration* in Jenkins job description.

Figure 2 - Specifying Integration tests to be executed in Jenkins

5. Integration Test Results

As integration tests are basically developed and executed in the same environment and using a similar set of tools – TestNG, Maven, Git, Jenkins - the presenting and interpreting test results are quite similar to the module tests described in deliverable 37.4 [3].

In this chapter we provide sample integration test results as they can be seen in Jenkins continuous integration environment of FutureID client testbed.

Following picture provides the trend view about the results of executed tests. On the chart green means passed tests and red is indicating failed tests. From this chart it can be seen that 80% of tests got fixed since build 21 and 20% of tests are still failing.

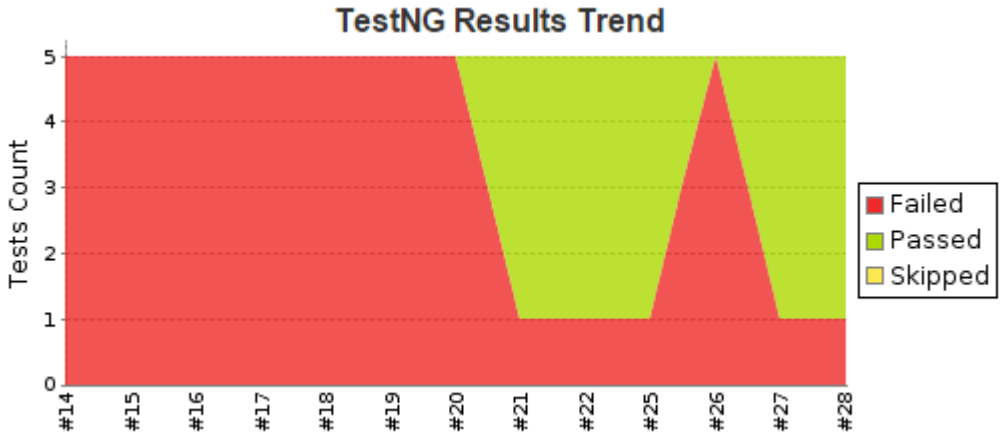

Figure 3 TestNG general view of PKCS11 Addon integration tests

Following picture provides detailed insight to tests executed during last build of integration test job.

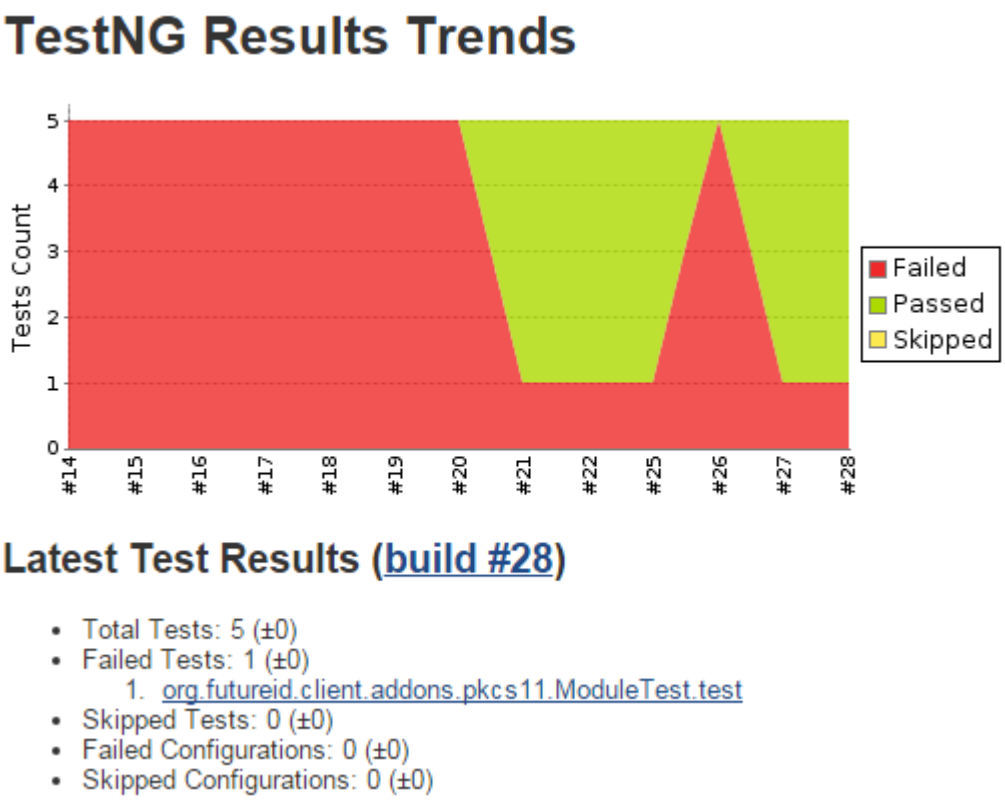

Figure 4 - Detailed view of TestNG results of one build

Package org.futureid.client.addons.pkcs11

1 failure(-4)

5 tests(±0)

All Classes

hide/expand the table

Class	Duration	Fail	(diff)	Skip	(diff)	Total	(diff)
AddonTest	00:00:01.568	0	-4	0	0	4	0
ModuleTest	00:00:00.005	1	0	0	0	1	0

Order of Execution by Test Method

hide/expand the table

Method	Duration	Start Time	Status
AddonTest.testFindObjects	00:00:01.568	Thu Oct 22 10:56:34 CEST 2015	PASS
AddonTest.testGetSlotList	00:00:00.021	Thu Oct 22 10:56:35 CEST 2015	PASS
AddonTest.testOpenSession	00:00:00.063	Thu Oct 22 10:56:35 CEST 2015	PASS
AddonTest.testReqID	00:00:00.009	Thu Oct 22 10:56:35 CEST 2015	PASS
ModuleTest.test	00:00:00.005	Thu Oct 22 10:56:35 CEST 2015	FAIL

Figure 5 Detailed view to tests of one package – PKCS11 Addon

6. Summary

Overall, the present report described integration testing in general and in the context of FutureID (§2). Moreover, it provided corresponding test tools that have smoothly been integrated into the testing approach and environment (§3). Further, the developed testing approach and infrastructure has been applied for the integration testing purpose. The obtained results have been presented in a concluding chapter (§5).

6.1 Conclusion

The client testbed has evolved over the entire duration of the project. Several intermediate steps have been achieved, which led eventually to a stable client testbed. It offers a testing environment that is suitable for module tests, acceptance tests and integration tests. The approach of annotating the tests as such has proved to be a good measure to keep track of all different kinds of tests and their corresponding results.

In agreement with the Server testbed partners, it has been decided to combine the Server and the Client testbed for the first time into one overall FutureID testbed. A significant advantage of this approach is, that dissemination and distribution of the FutureID testbed becomes easier and clearer, since the new potential user must be given only one instrument for the above mentioned multiple testing purposes.

6.2 Outlook

This overall FutureID test approach and environment can add value also in other research or consulting projects that have to use complex test scenarios combining many applications. It is highly envisioned to further operate and develop the FutureID testbed in various contexts. Since this developed testing approach is modular and open for adjustments and additions, it can easily be adapted to any kind of software testing settings. These steps are envisioned to be taken in future research and consulting projects and by integrating the FutureID testbed into the yet to be formed FutureID Foundation.

7. References

[1] S. Bradner, “Key words for use in RFCs to Indicate Requirement Levels,” 03 1997. [Online]. Available: https://www.ietf.org/rfc/rfc2119.txt. [Accessed 12 10 2015].

[2] E. Özmü, C. Ruff and J. Kubiziel, “Test strategy FutureID Client testbed,” 16 05 2013. [Online]. Available: https://dms-prext.fraunhofer.de/livelink/livelink.exe/properties/3002288. [Accessed 10 08 2015].

[3] J. Hofer, S. Trenton and M. Toom, “Acceptance Test Tools,” [Online].

[4] T. Libeskind and M. Toom, “Module Test Tools,” 2015. [Online]. Available: https://dms-prext.fraunhofer.de/livelink/livelink.exe?func=ll&objId=6417419&objAction=browse&viewType=1 . [Accessed 2015].

[5] “Maven - Project home page,” [Online]. Available: https://maven.apache.org/index.html.

[6] J. Hofer, T. Schulz and F. Lothar, “Test Assertions,” 22 04 2015. [Online]. Available: https://dms-prext.fraunhofer.de/livelink/livelink.exe?func=ll&objaction=overview&objid=5442590 . [Accessed 2015].

[7] “TestNG Project homepage,” [Online]. Available: http://testng.org/doc/index.html. [Accessed 2015].

[8] “TestNG Eclipse Plugin - Project homepage,” [Online]. Available: https://github.com/cbeust/testng-eclipse. [Accessed 2015].

[9] “Mockito - Project homepage,” [Online]. Available: http://www.mockito.org/. [Accessed 2015].

[10] T. Liebeskind and M. Toom, “First Internal Version of Client Test Bed,” 10 11 2014. [Online]. Available: https://dms-prext.fraunhofer.de/livelink/livelink.exe/properties/4787132. [Accessed 11 08 2015].