Markus Postl, BSc

Securing Lifecycle Tool Integrations

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Institute for Applied Information Processing and Communications

Head: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Supervisor: Dipl.-Ing. Dr.techn. Andrea Leitner (AVL)

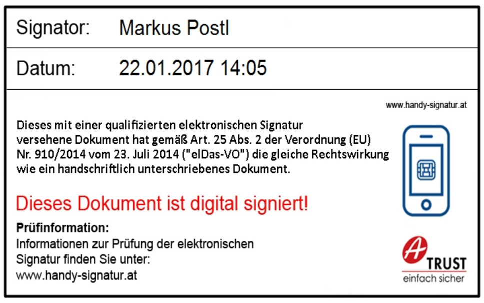
Dipl.-Ing. Bojan Suzic (IAIK)

Evaluator: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Graz, January 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.



Signator:	Markus Postl
Datum:	22.01.2017 14:05

www.handy-signatur.at

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr. 910/2014 vom 23. Juli 2014 ("eIDas-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

TRUST
einfach sicher

Date

Signature

Abstract

The growing complexity of application and product development processes and the related increase of incorporated tools pushes the development of standardized integration concepts. The Open Services for Lifecycle Collaboration (OSLC) is a set of specifications for coupling data beyond tool boundaries, based on technologies inspired by the web, such as RESTful services and Linked Data.

Making data accessible across tool boundaries induces new threats regarding application security. Hence, already at the stage of planning seamless toolchains, security can be considered as an integral part of the software concept.

In this work, security requirements for different scenarios of tool integration are elaborated. Especially authentication and trustworthiness between lifecycle tools, confidentiality, and integrity of the communication, as well as authorization to access resources, have been identified as important security requirements. Conventional security mechanisms are discussed in a threat and gap analysis, and the most promising mechanisms are integrated into the accompanying prototype implementation. The subsequent evaluation identifies security weaknesses of methods like Transport Layer Security (TLS), HTTP Basic Authentication, OAuth 1.0a, or OAuth 2.0. Findings of the prototype implementation and evaluation are discussed and supplemented by recommendations for the application of security mechanisms in different tool integration scenarios.

Consequently, OAuth 2.0 for delegated access control to resources, as well as a concept to establish trust relationships for server authentication are recommended. Finally, potential approaches for solving remaining open issues are presented.

Kurzfassung

Die steigende Komplexität in Anwendungs- und Produktentwicklungsprozessen und der daraus resultierenden Zunahme an eingesetzten Tools forciert die Entwicklung von standardisierten Integrations-Werkzeugen. Open Services for Lifecycle Collaboration (OSLC) ist eine Sammlung von Spezifikationen um Daten über Toolgrenzen hinweg miteinander zu verbinden. Die Spezifikationen basieren auf etablierten Internettechnologien wie REST und Linked Data.

Das Bereitstellen von Daten zwischen Tools induziert neue Bedrohungen im Bezug auf Anwendungssicherheit. Bereits bei der Planung von Toolketten ist Security folglich als integraler Bestandteil des Software-Konzepts einzustufen.

In dieser Arbeit werden Security-Anforderungen von verschiedenen Szenarien in der Tool-Integration erarbeitet. Als wichtige Elemente werden dabei die Security-Anforderungen Authentifizierung und Trust zwischen den Lifecycle-Tools, Vertraulichkeit und Integrität der Kommunikation, sowie Autorisierung zum Zugriff auf Ressourcen identifiziert. Konventionelle Security-Mechanismen werden in einer Threat- und Gap-Analyse diskutiert und die aussichtsreichsten in einer Prototyp-Implementierung umgesetzt. Die anschließende Evaluierung beleuchtet Schwachstellen von Methoden wie Transport Layer Security (TLS), HTTP Basic Authentication, OAuth 1.0a oder OAuth 2.0. Die Erkenntnisse aus der Implementierung des Prototypen und der Evaluierung werden anschließend diskutiert und Empfehlungen zum Einsatz von Security Mechanismen in verschiedenen Szenarien der Tool-Integration dargelegt.

Schlussfolgernd wird OAuth 2.0 für die Autorisierung zum Zugriff auf Ressourcen, sowie ein Konzept zum Aufbau von Vertrauensbeziehungen für die Server-Authentifizierung empfohlen. Abschließend werden mögliche Lösungsansätze für offene Punkte präsentiert.

Acknowledgement

I want to dedicate this page to all those people who supported me and made this thesis possible.

I am indebted to my supervisors Andrea Leitner and Bojan Suzic. I wish thanks to Andrea Leitner for proposing this work to me, introducing me to the topic, providing tools, reading and commenting the thesis, and above all for the organizational effort to even enable this work. I am very thankful to Bojan Suzic who guided me, pointed me to relevant work, backed me by carrying lots of organizational stuff, and supported me by reviewing and commenting on drafts of this thesis uncountable times.

Last but not least, I must express my gratitude to those close to my heart. I want to thank Isabella for supporting me during the last year, and for her steady understanding when I used a weekend again to write this thesis. I also want to thank my parents Erna and Josef for their continuous support in all matters.

Contents

List of Figures

List of Tables

Abbreviations

ABAC	Attribute Based Access Control
ALM	Application Lifecycle Management
API	Application Programming Interface
CA	Certificate Authority
CIA	confidentiality, integrity and availability
CSRF	cross-site request forgery
CRUD	Create, Retrieve, Update, and Delete
DAC	Discretionary Access Control
ESB	Enterprise Service Bus
IBAC	Identification Based Access Control
IdP	Identity Provider
IETF	Internet Engineering Task Force
IOS	Interoperability Specification
JWT	JSON Web Token
LDAP	Lightweight Directory Access Protocol
LTI	Lifecycle Tool Integration
MAC	Mandatory Access Control
MITM	Man-in-the-Middle
MOM	Message-Oriented Middleware
NIST	National Institute of Standards and Technology
OSLC	Open Services for Lifecycle Collaboration
PLM	Product Lifecycle Management

RBAC	Role Based Access Control
RDF	Resource Description Framework
RMIAS	Reference Model of Information Assurance & Security
SAML	Security Assertion Markup Language
SCIM	System for Cross-domain Identity Management
SDK	Software Development Kit
SOA	Service-Oriented Architecture
SSO	Single sign-on
SWT	Simple Web Token
TLS	Transport Layer Security
TRS	Tracked Resource Set
UMA	User-Managed Access
XACML	eXtensible Access Control Markup Language

1. Introduction

... Lifecycle Tool Integrations

A key issue in current computer-aided software engineering environments is the desire to link tools that address different aspects of the development process. [Was90]

The development process of products or applications requires the usage of several software tools. Typically, the data of those tools have mutual dependencies. A conventional approach to share data over tool boundaries is the usage of Application Programming Interfaces (APIs), or connecting tools via a common platform or enterprise bus system. Lifecycle Tool Integration (LTI) moves a step ahead and provides a unified way of linking information between all tools in the development process of a product. LTI additionally allows traceability of entities in the process. The Open Services for Lifecycle Collaboration (OSLC) [osl16] is an open community developing specifications for loosely coupled LTIs. The specifications are based on techniques inspired by the web, such as Linked Data, and RESTful services. The advantage of this concept is the idea to implement the LTI interface of a tool once, and use it multiple times to couple all tools in the lifecycle.

Securing ...

Security should never be an afterthought - it's an integral part of any software system design, and it should be well thought out from the design's inception. [Sir14]

Connected tools in a lifecycle widely open the doors for various threats by sharing data (resources), and by using common authentication and authorization flows. Hence, planning a concept for software security is an essential step, before tools are connected to share and link their resources.

Motivation and Goals. The OSLC specifications provide promising solutions for lifecycle tool integration. Accessing and linking data across tool boundaries requires methods for delegated access control and authorization. The specifications enumerate some security methods, but this list raises unanswered questions. How were these methods evaluated for security? Which methods should be applied for which integration scenario? Are there open issues which cannot be solved with these methods?

The goal of this work is to identify and evaluate solutions for defined integration scenarios. The elaboration of this goal was achieved step by step according to the structure of this work, as described in the next paragraph.

Structure of the Work. The thesis starts with a general overview of tool integration in Chapter 2. After describing the most important concepts for tool integration, the OSLC specifications are introduced. Finally, the integration scenarios considered in this work are defined at the end of the chapter.

Chapter 3 determines appropriate security requirements to the scenarios. Based on a reference model, a new model for the context of LTI is derived, and used to identify and clarify relevant requirements.

Chapter 4 outlines related work by introducing concepts and mechanisms used to achieve the security requirements. Further, the chapter lists relevant software solutions for LTI, including a proprietary software solution as well as an open source library.

The threat and gap analysis in chapter 5 analyzes threats and countermeasures in the defined scenarios for the introduced security methods as well as for existing software solutions. The chapter concludes with recommendations of methods for applying to the prototype implementation in the next chapter.

An architectural overview of the prototype implementation is given in chapter 6, as well as the data flow between tools including authorization. Extensive figures illustrate the data flow and explain the used security methods.

Chapter 7 pictures the outcome of this work with an evaluation of the prototype, as well as recommendations and future perspectives in the discussion.

The concluding chapter 8 briefly summarizes scenarios and security requirements and outlines the results from the prototype implementation and evaluation.

2. Tool Integration

Before we can discuss security in the area of LTI, we need to introduce fundamental tool integration concepts and scenarios. This chapter starts by introducing commonly used integration technologies, before the concept of OSLC is presented. Finally, integration scenarios using OSLC, which are deployed for implementation and analysis purposes in the next chapters, are introduced.

2.1. Integration Technologies

Wassermann [Was90] states that a key issue of a development process is to *"link tools that address different aspects of the development process."* This section gives an introduction to integration technologies which are commonly used and a demarcation to LTI. A more detailed overview of the described technologies is provided by [Men07].

2.1.1. Message-Oriented Middleware

In a Message-Oriented Middleware (MOM) architecture [Men07] all integrated tools are connected to a middleware layer, the message broker, pictured in Figure 2.1. Transmitted messages are decoupled, asynchronous and routed via the message broker. The middleware defines an interface for connecting all tools; the interface may be different for each tool. The message broker transforms the message to the data format of the receiving tool interface.

MOM solutions often use platform-specific and/or proprietary solutions which cause interoperability problems to alternative vendors [Men07]. This architecture is too inflexible for LTI.

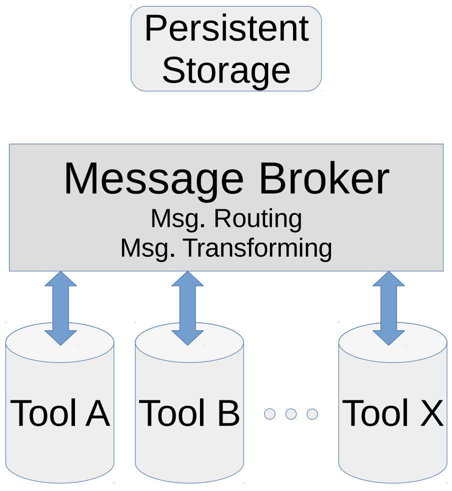


Figure 2.1.: Message-Oriented Middleware (MOM)

2.1.2. Service-Oriented Architecture

In a Service-Oriented Architecture (SOA), all tools provide their business logic as services [Men07]. Figure 2.2 shows a simplified version of SOA. Each tool registers its services and a description of the provided interface to a service registry. If *Tool A* requires some service, it first looks up the service registry to verify which connected tool provides the appropriate service. In a second step, it retrieves the service description from *Tool B* and consumes the described service. The communication typically is handled via SOAP or REST, [ECPB12] describes the concepts of those network communication methods.

Lifecycle tool integrations, in particular OSLC, use many of the approaches from the SOA architecture. Some of them are the usage of web technologies for communication, and listings of offered services by service registries/providers.

2.1.3. Enterprise Service Bus

An Enterprise Service Bus (ESB) is a message-based information infrastructure that provides interaction with distributed applications and services, in a secure and reliable manner, using open standards [Men07]. Typically an ESB is formed by a concatenation of service containers connected via a message bus, as shown in Figure 2.3. Each of the service containers provides different

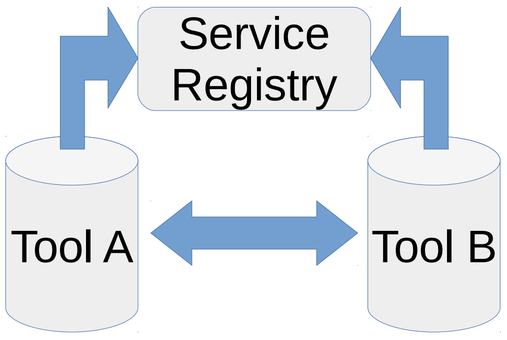


Figure 2.2.: Service-Oriented Architecture (SOA)

methods for integrating tools. There exist various software products, offering ESB capabilities; either free or commercial.

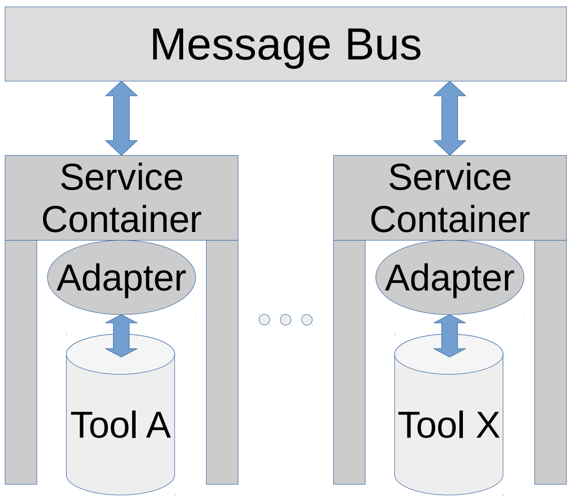


Figure 2.3.: Enterprise Service Bus (ESB)

This architecture is well suited in the enterprise scenario but requires custom adapters for each tool in the integrated lifecycle.

2.1.4. Conclusion

In this section, we introduced integration technologies. The MOM connects all tools via a middleware layer. The need to transform data between the different tools makes this solution inappropriate for LTI. An SOA allows tools to consume services of each other. The OSLC specifications use this technology to register and lookup services from other tools. The missing gap to the OSLC specification is the definition of the data format and possibilities to link data between different tools. Finally, we introduced the ESB. This solution builds on MOM and SOA and is typically used in enterprise scenarios. An example for an ESB in combination with OSLC is the IBM Jazz platform, which is described in Section 4.4.2. This thesis focuses on the OSLC specifications for tool integration, which will be introduced in the next section.

2.2. Open Services for Lifecycle Collaboration

This section briefly introduces a set of specification for tool integration, developed by the working groups of the OSLC initiative [osl16], initially started in 2008, and governed by the steering committee [1] at the OASIS Open Standards Network. The goal of the OSLC initiative is to make software that integrates easily with other software for building connected development environments.

OSLC

Figure 2.4.: Logo of the Open Services for Lifecycle Collaboration

In contrast to the integration technologies discussed in Chapter 2.1, the specifications define how to link data seamlessly between tools without copying the information, or extracting it to a common database. The model can be used either for Application Lifecycle Management (ALM) or Product Lifecycle Management (PLM).

The base concept is linked data; using web technologies such as URI and the Resource Description Framework (RDF) [MMM+04], which is a data model

[1]http://www.oasis-oslc.org/governance (Accessed: 2016-12-28)

without type for describing and modeling of information. An artifact (e.g. a requirement) can be uniquely identified and therefore accessed by other tools. RESTful services allow to retrieve and search artifacts from tools with HTTPs Create, Retrieve, Update, and Delete (CRUD) operations.

2.2.1. Core Specification

The OSLC core specification [osl13] is currently in final version 2.0. Version 3.0 is in draft state. In the following, always the final version 2.0 is mentioned.

Figure 2.5 illustrates the concepts and relationships of the OSLC core specification. It defines the basic means of communication between tools. The *Service Provider Catalog* lists *Service Providers* and may include OAuth configuration and publisher information. A *Service Provider* facilitates a *Creation Factory* and a *Query Capability*. The *Creation Factory* can be used to create new resources using `HTTP POST`. A *Query Capability* lists the URIs of contained resources. The resources can be retrieved, updated or deleted by using `HTTP GET`, `HTTP POST`, or `HTTP DELETE` on the resource URIs.

2.2.2. Authorization

To control access to resources from an OSLC service provider the core specification defines an *OAuthConfiguration* resource to store three URLs for the use of OAuth.[OAu09] Furthermore, three different ways of authentication are recommended:

- HTTP Basic Authentication via SSL (TLS)
- OAuth Authentication
- Form Based Authentication

The specification does not mandate to use one of the above methods. In draft version 3.0, additionally, OAuth 2.0 and OpenID Connect is recommended. These and further security methods are described in Section 4.3.

2.2.3. Domain Specifications

In addition to the core specification, there are several domain-specific specifications, e.g. for change management, requirements management, and quality

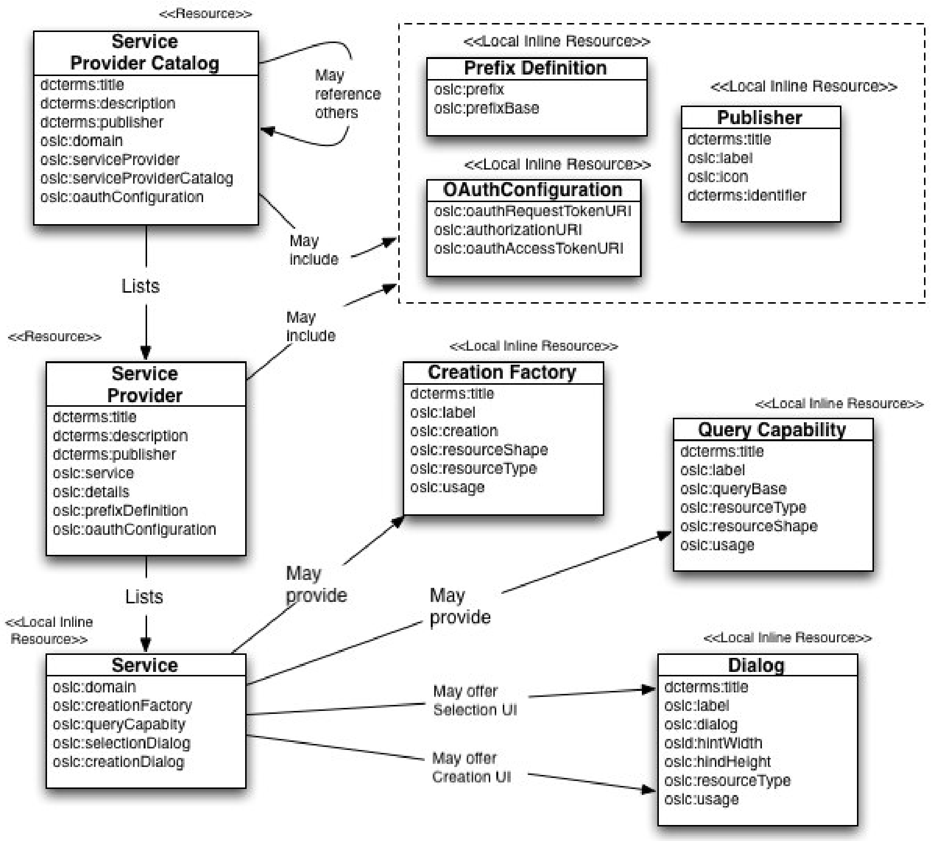


Figure 2.5.: OSLC conceptual model from [osl13]

management. The domain specifications define the basic terminology required within these domains.

The example OSLC scenarios in this work and the accompanying prototype implementation are in the domain of requirements management. The OSLC requirements management specification [osl12] defines requirements as *"basis for defining what the system stakeholders (users, customers, suppliers and so on) need from a system and also what the system must do in order to meet those needs, and how the surrounding processes must be orchestrated so that quality, scope and timescale objectives are satisfied."* A framework for testing in the domain of requirement management which integrates tools via OSLC is presented by Aichernig et al. [AHL$^+$14]

2.2.4. Tracked Resource Set

The specification for *Tracked Resource Sets (TRSs)* [osl15b] can be used in extension to the OSLC core and domain specifications. Currently, the specification is in finalizing draft status. It defines a protocol which allows a server to expose a set of resources. Clients can discover resources in the set, and track modifications of the set.

A use case of the protocol is a centralized service extracting data from multiple tools via TRS, and allowing clients to search in an indexed database of extracted data. This enables tools to search for data in a more convenient way, e.g. with SPARQL queries. SPARQL [PSH08] is a query language for RDF resources.

An example data flow is shown in Chapter 6.

2.3. Integration Scenarios

Two integration scenarios will be considered in this thesis to evaluate different security methods for LTI. The scenarios will be used in the subsequent chapters for analysis of threats, implementation of a prototype, and evaluation of the implementation and used security methods.

The first scenario handles the direct connection of tools via OSLC; the second scenario additionally handles TRSs.

2.3.1. OSLC Scenario: Direct Communication via OSLC

The OSLC scenario describes a toolchain, loosely coupled via OSLC. Figure 2.6 describes an example lifecycle in the automotive sector with the successive steps. The deployed tools of this example toolchain are not relevant for this work and will not be further described.

1. Requirements are defined in a requirements management tool.
2. A system specification is created via a SysML tool based on the defined requirements.
3. The specification is handed over to a simulation tool, e.g. AVL Model.CONNECT.
4. Export the simulation results to a data management platform, e.g. AVL SANTORIN.

5. (Optional) Read the requirements from the requirement management tool, refine (modify) them and update the requirements in the requirement management tool, e.g. by VIRTUAL VEHICLE REFINE.
6. Import the requirements and the stored simulation results to a validation tool, e.g. AVL MAGIC evaluates the result and AVL VeVaT checks if all requirements are fulfilled.
7. Store the validation results to the requirements management tool.
8. (Repeat) Further refine requirements or system specification.

The scenario investigated here focuses on the communication parts via OSLC; the other parts are usual API communication protocols which are out of scope in this work. The relevant parts of the OSLC scenario are marked in Figure 2.6 with a grey box. Thus are the communication parts between Refine, the requirement management tool, and the validation tool (AVL VeVaT). The connections between those tools implemented using OSLC. A similar scenario which integrates Refine, VeVaT, and a requirements management tool using OSLC was introduced by Marko, Leitner, Herbst & Wallner [MLHW15].

The challenges of this scenario are to analyse if the security mechanisms, proposed by OSLC, fit all security requirements, and discuss other relevant security and access control mechanisms.

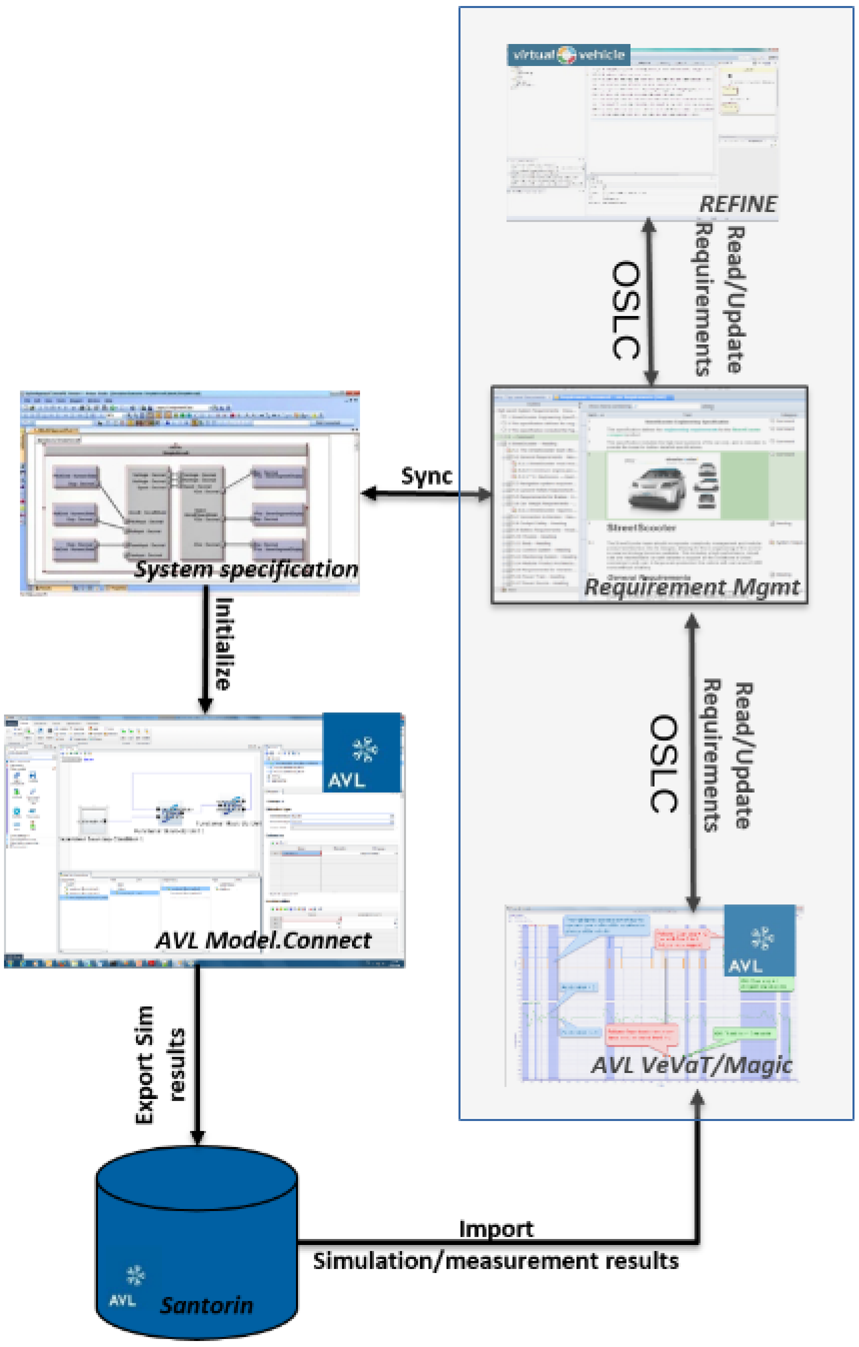


Figure 2.6.: OSLC Scenario - Direct Communication via OSLC

2.3.2. Platform Scenario: Communication via Centralised Integration Platform

The Platform Scenario is more complex as it defines not a fixed model, but an architecture for integration.

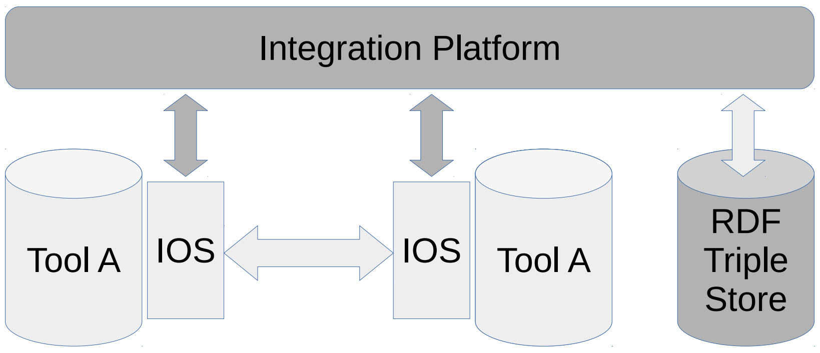


Figure 2.7.: Platform Scenario - Communication via a Centralised Integration Platform

As clearly represented in Figure 2.7, on top of the system is a centralized integration platform. The tools may communicate directly via an Interoperability Specification (IOS), e.g. OSLC, or communicate via the integration platform. In addition to the peer-to-peer system of the OSLC scenario, the integration platform can be used to handle authentication and access control. Furthermore, centralized databases, e.g. RDF triplestores, extract and permanently store resources from different lifecycle tools. This provides additional features to the users. First, it allows to access information from tools, even if they are offline. And second, information from different tools can be searched or queried easily with query languages like SPARQL.

An additional challenge to the OSLC scenario may be to enable a Single sign-on (SSO) experience to the user. The user should not be forced to log into each single tool, but only to the integration platform. A more complex challenge is access control. As long as the resources are stored within one tool, the tool can control access to the resources. But after extracting resources by the integration platform to a database, the integration platform obtains responsibility of access control to the resources. Access control policies need to be centralized or kept synchronized with the individual tool policies.

3. Security Goals and Requirements

Defining security goals[1] is crucial before we can discuss the requirements[1] of different scenarios of LTI. A common approach of defining security goals is the triad of confidentiality, integrity and availability (CIA) [WM11]. But the CIA triad is far away from a complete list of security goals, threats have evolved, and extended models became necessary. We decided to derive our requirements on a more recent model, the Reference Model of Information Assurance & Security (RMIAS) [CH13], considering the context of LTI.

The following Section 3.1 describes the security goals introduced as part of the RMIAS model. In Section 3.2 we define our derived security requirements for LTI. Section 3.3 lists non-functional requirements which have to be considered when applying methods for achieving security requirements.

3.1. RMIAS Security Goals

The Reference Model of Information Assurance & Security was proposed by Cherdantseva and Jeremy [CH13] to address the evolution of security threats by providing an abstract framework for information security. It is a model with four dimensions: Information System Security Life Cycle, Information Taxonomy, Security Goals and Security Countermeasures. The dimension of Security Goals contains the following goals:

- *Accountability*
- *Auditability*
- *Authenticity/Trustworthiness*

[1]NOTATION HINT: Security goals in the context of this work are abstract security-related targets. Security security requirements define the set of mandatory requirements of this work.

- *Availability*
- *Confidentiality*
- *Integrity*
- *Non-repudiation*
- *Privacy*

In the remainder of this section, we give a short explanation of each security goal and put them in the context of LTI.

3.1.1. Accountability

Definition from RMIAS: An ability of a system to hold users responsible for their actions (e.g. misuse of information).

The system needs to know in a transparent way which information is provided to whom and which modifications have been made. Typically a user is responsible for an action, in the context of LTI accountability may be additionally relevant on tool level, e.g. which tool accessed information from a service provider.

According to Pearson and Charlesworth [PC09], accountability compromises the following claims:

- Transparent usage of data.
- Assurance of privacy policies.
- Trusting users, by providing information on the usage of user's data.
- Responsibility, by implementing diligence and compliance measures to conform to regulations.
- Policy compliance to ensure services comply with laws and organisational policies.

3.1.2. Auditability

Definition from RMIAS: An ability of a system to conduct persistent, non-bypassable monitoring of all actions performed by humans or machines within the system.

Typically each tool implements its own system to monitor actions within the tool, as well as the incoming and outgoing communication. The missing gap in the integration scenario is what happens between the tools, supposed the tools don't communicate directly with each other.

3.1.3. Authenticity/Trustworthiness

Definition from RMIAS: An ability of a system to verify identity and establish trust in a third party and in information it provides.

In tool integration, a tool A depends on information given by a tool B. For A *trustworthiness* of B and *authenticity* of the information provided by B is mandatory. Thus, we need mechanisms allowing A establish trust to B and mechanisms to validate B is the originator of received messages. *Authenticity* is about verifying the originator of received messages, and *trustworthiness* is related to the establishment of a trust relationship between tools.

3.1.4. Availability

Definition from RMIAS: A system should ensure that all system's components are available and operational when they are required by authorized users.

Availability targets to the ability of providing information and services if they are required. It heavily depends on the architecture of a system. A failure of a component which is required by many other components, a single point of failure, is critical to a distributed system. In the scenario of LTI, at least centralized parts as authentication services or service registries need to be secured towards *availability*. But tool integration generates further questions: how can we provide that the information hold by a distinct tool is available when required.

3.1.5. Confidentiality

Definition from RMIAS: A system should ensure that only authorized users access information.

Each tool needs to protect its data, no matter if it is stored in a tool's database or transferred between tools. Providing *confidentiality* depends on other security goals as authenticity or trustworthiness. In the context of LTI, the protection of the communication between trusted tools from unauthorized parties is essential.

3.1.6. Integrity

Definition from RMIAS: A system should ensure completeness, accuracy and absence of unauthorized modifications in all its components.

In addition to authenticity, a tool needs to ensure *integrity* of information to guarantee that the information was not altered. In tool integration we need to protect all data extracted from a tool. Therefore we need trust relationships between all communication partners and mechanisms to check received messages were not altered on the communication channel.

3.1.7. Non-repudiation

Definition from RMIAS: An ability of a system to prove (with legal validity) occurrence/non-occurrence of an event or participation/non-participation of a party in an event.

A mechanism providing *non-repudiation* needs to ensure none of the communication partners exchanging messages can deny participation in a part or the whole communication [KMZ02]. Therefore evidences of the origin and receipt of messages need to be done. An interpretation of the above definition is that the integration system requires to be able to prove who sent which messages between two tools. To allow such a proof we rely on some of the other security requirements, as *integrity*, *authenticity* and *trustworthiness.*

3.1.8. Privacy

Definition from RMIAS: A system should obey privacy legislation and it should enable individuals to control, where feasible, their personal information (user-involvement).

The protection of information from unauthorized parties is given by the *confidentiality* goal. The information an user is allowed to see is derived in the ideal case from the integrated tools. The same holds for personal information of an user. But it is important to track the information flows and specify which information is exposed to whom.

If personal information (e.g email address, or user name) is necessary for tool integration, it is required to protect those information from unauthorized parties.

3.2. Security Requirements

The following definition of security requirements holds for both scenarios, the OSLC Scenario described in Section 2.3.1 and the Platform Scenario described in Section 2.3.2. Not all of the requirements might be necessary for every use-case of the scenarios. Especially in a trusted secure environment, such as a company intranet, some of the requirements might be from less interest than in an insecure environment or communication via the Internet.

Security goal have been prioritized. Only the most important once will be considered in the following. From the other security goals presented in the previous section, security requirements are derived.

The following RMIAS security goals will not be discussed in this work:

- **Accountability:** Identifying the responsibility for misuse of data and controlling compliance on regulations is an important security goal. But installing a system for accountability is not part of this work.
- **Auditability:** In a first step it is important to provide a secure communication between tools and/or a secure integration platform. The security goal of auditability needs to be discussed in future work.
- **Availability:** Providing information and services when they are required is crucial for every distributed system. Mostly this can be achieved by an appropriate architecture and IT infrastructure. This work focuses on other security goals and availability is not specified as security requirement.
- **Non-repudiation:** Proving the receipt and the originator of messages may be a legal requirement for some specific use-cases. For our scenarios non-repudiation was identified as out-of-scope.
- **Privacy:** Protection of personal information of an user is more than encrypting the communication channels. Even in a trusted relationship of two tool, one tool might not handle sensitive data over to the other

tool. Different methods can be used to protect those data, including access control methods, cryptography, and masking or transforming of data. In this work privacy is not discussed, except the protection of user credentials.

The following sections identify the security requirements and describe the relation to LTI.

3.2.1. Authentication

This requirement covers the security goals *authenticity* and *trustworthiness*. For the LTI scenarios the requirement can be further partitioned:

1. **Authentication of the communication partner**. A tool shall accept only messages from a trusted party. In an insecure environment this deduces the need to authenticate the communication partner, another tool or the integration platform.
2. **Authentication of the user**. A tool may need to authenticate an user before it provides access to a resource. This leads to the requirement of user authentication and federated user identification to establish a mapping between multiple tool identities of a particular user.

3.2.2. Authorization and Confidentiality

The authorization requirement is deduced from the security goal *confidentiality*. The system shall grant only authorized parties access to a resource. This implies to protect the resources with an access control method. In our scenarios the access control can be handled by the tools or the integration platform. Furthermore this requirement demands, data sent via insecure communication channels shall be protected from all types of unauthorized disclosure or modification.

User credentials required for authentication/authorization need more protection than just encrypting the communication channels. Credentials required to authenticate to a service provider shall not be accessible by the client application. This is a consequence of missing trust between tools. A tool may not assume, another tool is able to protect user credentials in an appropriate way.

3.2.3. Integrity

Integrity of data in transit shall be ensured by the client tool. This requirement is based on a previous authentication and trustworthiness of the communication partner. A communication partner may be a third party tool or components of the integration platform.

3.3. Additional Non-Functional Requirements

The ISO/IEC 25010:2011 [ISO11] defines *security* as a non-functional requirement. This section introduces additional non-functional requirements contiguous to the security requirements. The additional non-functional requirements have to be considered when security methods are applied. Some security methods may fully comply with all security requirements, but might be inadequate to meet the additional non-functional requirements.

3.3.1. Usability

Adding a security layer to LTIs shall not affect the usability of the system in a way the users are displeased. All unnecessary interaction of the user with the system needs to be avoided. Multiple logins to different tools or often repeating logins by the user shall be avoided.

3.3.2. Simple Configuration and Administration

Administration covers activities as the installation of tools, creation, and deployment of security certificates, or user/role management. The initial configuration and the continued administration of security related components shall be of reasonable effort for the system administrator. The security layer shall not increase the effort of administration in an inappropriate way.

3.3.3. Interoperability and Lightweight Integration

Exclusively, standardized methods shall be used to increase interoperability of tools. Further, the security layer shall not increase the complexity of the LTI system in a way that costs and effort to adapt the tool for the integration are inappropriate.

4. Related Work

This chapter gives an overview of related work by introducing concepts and mechanisms which are used to fulfil the defined security requirements for the previously defined scenarios. Further, software solutions for LTI are introduced.

The chapter starts by introducing Levels of Trust as a proposed solution to specify trust relationships between tools for particular use cases. Access control is part of the security requirements, therefore the most common access control concepts are described. In Section 4.3, a list of security mechanism related to the LTI context are discussed. At least, existing software solutions for LTI are described.

4.1. Levels of Trust

In an integration scenario with multiple tools, the trust relationships between those tools may differ. E.g., *trustworthiness* of a tool might be assumed in a closed intra-corporate network.

We need a way to specify those varying degrees of trust. Therefore, we propose an approach which is derived from X.509 certificates chains [CSF$^+$08], used in public key infrastructures. [GIJ$^+$12] describes the *chain-of-trust* concept used for X.509 certificates. The concept utilizes levels of trust to verify the validity of a certificate.

In the LTI scenario, the first outer level could contain tools connected across corporation borders (inter-corporation). But also within a corporation we can distinguish between projects, departments, and other classifications.

These levels lead to an onion-like structure, which differs from every integration scenario.

Abstract example of levels of trust in a corporation scenario: Figure 4.1 is an abstract example of an integration scenario with four different levels

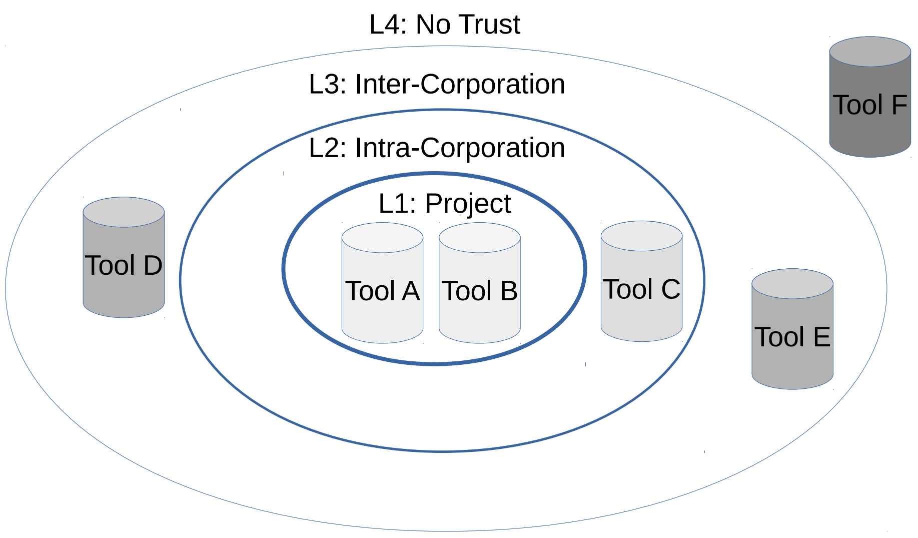


Figure 4.1.: Levels of Trust in an onion like structure

of trust. This levels have to be modified and/or expanded depending on the use-case.

- **L1: Project:** Full trust between tools
- **L2: Intra-Corporation:** Trustworthiness of tools, but no assumptions of confidentiality and integrity of the intra-tool communication.
- **L3: Inter-Corporation:** No assumptions on trustworthiness, confidentiality, or integrity of communication, but those can be assured with according security mechanisms.
- **L4: No Trust:** Impossible to guarantee correctness of the information from those tools.

4.2. Access Control

Access control [Kha12] is generally a mechanism or procedure that allows, denies, or restricts access to a system.

In the field of LTI there are a couple of issues related to access control that need to be addressed:

- **Identification of Users.**
  A mechanism granting access to a system requires the previous knowledge

of the user making the access attempt. Thus, the problem of access control implies the problem of user identification. All security goals related to user identification, described in Section 3.1, need to be fulfilled before access control can be provided. To identify the user we need *authenticity/trustworthiness*, *availability*, *confidentiality*, and *integrity*. In a company scenario, the identity of an user may be provided in form of authentication assertions by an identity management system. An attribute based access control model which requires no user identification is unlikely in LTI, since most tools are build upon user or role based authentication.

- **Mapping of Tool Users.**
  A typical approach for access control by lifecycle tools is some kind of access control based on users and/or roles, corresponding to an Role Based Access Control (RBAC) model, which is introduced in the next section. All tools, which are deployed for the prototype implementation in Section 6, have access control models based on RBAC. The integration system needs to know mappings of users and roles between different tools.
- **Diversity of Access Control Models.**
  Various methods providing access control exists; an overview is given in Section 4.2.1. This infers the problem of providing a suitable federated access control system.

4.2.1. Methods of Access Control

A simple form of access control is to allow access to a resource based on the user's identity. This approach is called Identification Based Access Control (IBAC). But it is not scalable enough for the integration scenario. Various models [MNN14] exist in extension to IBAC. This section describes some of the most frequently used models.

Mandatory Access Control (MAC) is the most important access control model to military applications [MNN14]. MAC is a centralized model, based on security enforcement rules defined by an administrator. Rules cannot be deleted, updated, or added by an user. The decision to grant access to a resource is done by identifying the user and the resource, and deciding based on the defined security level of the user and the sensitivity of the resource. The main advantage of MAC is simplicity and control of the system by an administrator [MNN14].

A disadvantage of the model is its inflexible behavior. In an integration scenario with frequently changing tools, users, and resources, keeping rules up-to-date defining by an administrator is an expensive task.

Discretionary Access Control (DAC) is the most widely used model of access control [AC01]. In difference to MAC, not all access rights are predefined and can be modified by the holder of the resource. Furthermore, users can form groups, and rights can be assigned to groups instead of users. This model enables a fine-grained access control, but maintenance and verification of rights is difficult as the user controls the rights.

Role Based Access Control (RBAC) [FK92] can be seen as a combination of MAC and DAC. Rights are predefined by an administrator and can not be modified by the resource owner. In difference to MAC and DAC [AC01], rights are assigned to roles instead of users or groups of users. In a second step, one or multiple roles can be assigned to users. Thus, the resulting model has the advantages of MAC combined with the possibility of fine-grained access control. However, the problem of administrating large systems remains.

Attribute Based Access Control (ABAC) [Kha12] creates decision based on attributes. Access rights are granted by a combined validation of attributes from the requestor (e.g. role, job title), the service (e.g. read, write), the resource being accessed, and the environment (e.g. time, location). Thus, an ABAC system is composed of four entities:

1. **A requestor** sends requests and invokes actions to the service.
2. **A service** providing an interface with pre-defined operations.
3. **A resource** shared among different services, with a specific set of state data.
4. **An environment** contains information that might be useful for making the decision, such as date and time.

Each resource can be associated with a set of attributes; the access structure of an user is defined as a logical expression over these attributes [MNN14]. The model allows a fine-grained, and scalable access control.

4.3. Security Mechanisms

In the previous chapters, we defined integration scenarios and security requirements. This section gives an overview of frequently used security mechanisms and protocols in the context of the security requirements.

4.3.1. Transport Layer Security

Provides: *Authentication* and *Integrity*.

The TLS protocol [DR08], formerly Secure Sockets Layer (SSL), is a standard of the Internet Engineering Task Force (IETF)[1] providing communication security. It is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol.

The TLS Record Protocol uses symmetric encryption for data encryption, to guarantee that the connection is private. Keyed-Message Authentication Codes (MAC) are used for integrity checks to guarantee the connection is reliable.

The TLS Handshake Protocol provides mechanisms for authentication using asymmetric or public key cryptography. The authentication is optional but recommended for one or both communication partners. The protocol guarantees that the negotiation of a shared secret is secure and reliable.

There are two different configurations of TLS, one-way with server certificates and two-way *(mutual TLS)* with server and client certificates. Mutual TLS additionally allows the server to authenticate the client. In LTI mutual TLS is not supported in general, as the assumption of existent and valid client certificates does not always hold.

Adding security via TLS can be easily implemented by the usage of the web protocol HTTPS instead of HTTP.

4.3.2. HTTP Basic Authentication

Provides: *Authentication*.

HTTP Basic Authentication [FHBH+99a] is part of the HTTP protocol [FHBH+99b] and specified by the IETF for the purpose of user authentication. The authen-

[1]https://www.ietf.org (Accessed: 2017-01-06)

tication is handled via the HTTP Authorization header. The web server replies an unauthenticated request from a client with the header:

```
WWW-Authenticate: Basic realm="RealmName"
```

The client's browser tries to authenticate the requested realm. If there is no open session, the user is asked for username and password in a dialog box. Subsequently, the username and password are send encoded, but not encrypted, to the web server via the HTTP header:

```
Authorization: Basic base64encoded=un&pw
```

To protect username and password, and guarantee *authenticity* of the server, typically HTTPS is used.

The great advantage of these protocol is its simplicity and high degree of popularity. But there are some shortcomings. At first, the HTTP Authorization header needs to be transferred with every request, resulting in a lower performance for authentication checks and the requirement to use TLS always. Furthermore, if we login to a third party tool, the browser login box is decorated out of place with a different look and feel. At last there is no way to provide a logout, the session is handled by the browser or browser platform of the requesting tool.

4.3.3. Form Based Authentication

Provides: *Authentication*

Form based authentication is no specification, it describes a method of authentication using a form. Typically a user has to enter username and password at a web page. However, for needs of LTI, redirection to an authentication page is not sufficient. The service provider needs to inform the client tool about a successful login, and the service provider may need to authenticate the client tool as well.

A possible solution is to use a web API for login instead. Therefore, the client sends the user credentials with a HTTP request to the service provider. This solution carries along similar problems as HTTP Basic authentication. The user credentials are exposed to the client and sent in plaintext to the service provider.

4.3.4. Security Tokens

Security tokens [Rou12] are often used to pass information between communication partners. All of the following protocols use some sort of security tokens.

A very simple form are **Simple Web Tokens (SWTs)**. These tokens consist of name/value pairs, called attributes. The HMAC-SHA256 attribute is always the last and mandatory, it is a `HMAC-SHA256` keyed-hash of all other attributes.

A **JSON Web Token (JWT)** is a method for representing claims, encoded as JSON objects, defined by the IETF [BSJ15]. The objects are encoded, digitally signed and optionally encrypted. JWTs are separated to three parts: JWT header, JWT second part and JWT third part. The header describes the cryptographic operations within the token, the second part is used as payload or encrypted key. The third part is reserved for the signature or ciphertext. A JWT contains, quite similar to SWTs, name/value pairs in form of a JSON string set. Each of the pairs is called claim, each claim name is unique within a JWT.

4.3.5. OAuth

Provides: *Authorization*

OAuth [Rou12] is specified in multiple versions, this section describes OAuth 1.0a [OAu09]. The purpose of this protocol is to enable consumers (applications) to access resources from a service provider, without requiring the user to disclose authentication information to the consumer application.

OAuth defines three roles: *consumer*, *service provider* and *user*. In a typical web application scenario, the consumer is the user's web browser, the service provider is a web application allowing to access protected resources via OAuth, and the user is the owner of the resources.

OAuth defines three request URLs: The *Request Token URL* is used to obtain an unauthorised request token, the *User Authorization URL* is used to obtain user authorization for consumer access, and the *Access Token URL* is used to exchange the user-authorised request token for an access token.

The OAuth protocol parameters are sent encoded (and signed) in the HTTP Authorization header.

The OAuth workflow consists of three steps:

1. The consumer obtains an unauthorized request token from the service provider.
2. The user authorizes the request token. Therefore the user is redirected to the service provider and enters authentication information.
3. The consumer exchanges the request token for an access token from the service provider.

After a successful authorization, the consumer can access protected resources from the service provider with the access token. All request must be signed from the consumer and verified by the service provider. The protocol defines the signature methods `HMAC-SHA1`, `RSA-SHA1`, and `PLAINTEXT` (only if the communication is protected via HTTPS).

Table 4.1.: OAuth 1.0 vs OAuth 2.0 Terminology

OAuth 1.0	OAuth 2.0	Description
User	Resource Owner	The owner of a resource, typically a user (person).
Service Provider	Resource Server	A service hosting resources.
Consumer	Client	An application accessing resources of the service provider.
Request Token	Authorization Code/Grant	A value expressing authorization from the user, can be exchanged for an access token.
Access Token	Access Token	A value used to gain access to a protected resource.
Consumer Key	Client ID	A value used to identify the Consumer to the Service Provider
Consumer Secret	Client Secret	A secret to prove ownership of the Consumer Key.

4.3.6. OAuth 2.0

Provides: *Authorization*

The OAuth 2.0 Authorization Framework is based on OAuth 1.0 and is standardized by the IETF [Har12]. It allows a third-party application to obtain access to a service. Table 4.1 gives a list of important terminology differences between OAuth 1.0 and OAuth 2.0.

OAuth 2.0 defines four roles: the *resource owner* is capable of granting access to a protected resource, it is called end-user if the resource owner is a person; the *resource server* is hosting the protected resources and is capable of answering resource requests using access tokens; the *client* is an application which is requesting protected resource on behalf of the resource owner; and the *authorization server* is authenticating the resource owner and issuing access tokens to the client.

The OAuth 2.0 workflow consists of three steps:

1. The client requests authorization from the resource owner, and receives an authorization grant. The authorization is preferably done indirectly via the authorization server.
2. The client requests an access token from the authorization server by presenting the authorization grant.
3. The client requests a protected resource from the resource server by presenting the access token.

OAuth 2.0 introduces the term *bearer token*. Using bearer tokens is one possible way how to request tokens; it is the simplest way and specified as the default. Any party in possession of a bearer token can use the token. It does not require to make a proof-of-possession, e.g. by the usage of cryptography.

From the steps above we see that OAuth 2.0 has a cleaner separation of roles, especially the separation of resource server and authorization server. Furthermore, signatures from OAuth 1.0 are obsolete; to protect requests, OAuth 2.0 recommends HTTPS. The protocol is designed to provide better support for non-browser based clients. Refresh tokens are introduced, which allow generating new access tokens without the need to follow all of the three steps. This simplifies the procedure of creating short-lived access tokens (session tokens) and long-lived refresh tokens. Finally, a series of other standards is based on the OAuth 2.0 standard, including UMA and OpenID Connect.

4.3.7. OpenID Connect

Provides: *Authentication*

The OpenID Connect protocol [SBJ+14] is an identity layer on top of OAuth 2.0, using bearer tokens and JWTs. The purpose of the protocol is to provide *authentication* based on OAuth 2.0.

OpenID Connect introduces an *ID token*, a JWT containing claims about the authentication of an end-user. The token must include information about the issuer, the subject (an identifier for the end-user), the audience(s) the token is intended for, the expiration time and the issue time, as well as optional and custom claims.

The OpenID Connect workflow consists of the following steps (notations from OAuth 2.0 are used):

1. The client sends an authentication request to the authorization server. The authorization server authenticates the end-user and redirects the end-user back to the client with an authorization code.
2. The client sends a request, using the authorization code, to the token endpoint. The token endpoint is a service used to exchange authorization token for a access token, or ID token in OpenID Connect. The token endpoint responds with an ID token and access token.
3. The client validates the ID token and retrieves the end-user's subject identifier.
4. The client requests protected resources using the access token or retrieves information the the end-user using the UserInfo Endpoint. The UserInfo Endpoint returns claims about the end-user.

4.3.8. Security Assertion Markup Language

Provides: *Authorization* and *Authentication*

Security Assertion Markup Language (SAML) is a set of specifications for federated identity management [BT11]. SAML is standardized by the OASIS[2] standards consortium in the current version V2.0 [oO05]. Identity Federation [MKL09] allows to link different identities of a subject, which are managed by different service providers. A subject is typically an end-user. The end-user is authenticated to an identity provider, and the information is shared between the identity provider and the service providers. Therefore one major application of SAML is SSO. All requests and responses of SAML are done with XML.

The specification is composed of four main components: assertions, protocols, bindings, and profiles.

- **Assertions** express security information about subjects, used by service providers to make decisions. There are three types of assertions: Authentication, attribute, and authorization decision. An *Authentication* statement describes when and how the subject was authenticated to the identity provider. The *attribute* statement describes the attributes of a subject, e.g. the name of the user. Finally, the *authorization decision* statement expresses which resources the subject is permitted to access under which conditions.
- **Protocols** define the request and response pairs of SAML messages. The protocols are defined independently from the used communication protocols. The SAML core specification includes protocols for assertion query, authentication, SSO and other.
- **Bindings** define the mapping between the SAML protocol messages and the used communication protocols. The most important bindings are to `SOAP`, `HTTP GET`, and `HTTP POST`.
- **Profiles** There are various core protocols and some protocols for the use with other specifications, e.g. XACML. In general profiles build the top layer of the SAML structure. Each profile targets a specific function, e.g. SSO. Different combinations of protocols and bindings allow different implementations of the same profile.

[2]https://www.oasis-open.org (Accessed: 2017-01-06)

The functionality of SAML can be compared to OpenID Connect providing authentication and authorization. For comprehensive authorization systems SAML recommends profiles which combine SAML with XACML. The advantages of SAML are its specification which allows various different implementations and the high acceptance of the standard, especially in the enterprise segment. On the other hand it is more complex then token based systems as OpenID Connect or OAuth.

4.3.9. User-Managed Access

Provides: *Authorization*

User-Managed Access (UMA) is a specification [HMMC15] recommended by the Kantara Initiative[3]. The specification is a profile of OAuth 2.0 and defines how resource owners can control access to protected resources by arbitrary clients. The authorization is governed by a centralized authorization server based on policies of the resource owner.

The UMA protocol composes three phases, (notations from OAuth 2.0 are used):

1. **Protect a resource:** The resource owner introduces the resource, protected at resource server, to the authorization server. The authorization server starts to protect the resources based on OAuth. UMA does not specify how the resource owner configures the authorization server with policies for the protection of the resources.
2. **Get authorization:** The client wants to access a protected resource of a resource server. The client must first gain authorization from the authorization server. The API is protected based on OAuth.
3. **Access a resource:** The client successfully presents the resource server the authorization gained in the second step and obtains access to the protected resource.

The UMA specification recommends *OpenID Connect* for authentication if an identification of the end-user is required in addition to the authorization needs provided by UMA and OAuth 2.0.

[3]https://kantarainitiative.org (Accessed: 2017-01-06)

The possibility of centralizing and federating the authorization is an interesting concept for tool integration. It's OAuth 2.0 based design makes the concept flexible for different use-cases. Currently, UMA is an RFC draft and is in the process to get standardized by IETF. Such a standardization would probably lead to an higher acknowledge and acceptance by vendors. Beside the publicity of the specification its complexity is another disadvantage. It is considerably more complex than OAuth or OpenID Connect.

UMA will not be further discussed in this work. At the time, there are hardly any real world implementations. Hence, either any tools support those method, why the effort for implementation would be increased compared to other methods. For further reading, [Suz16] analyzes and compares OAuth 2.0 against UMA for cloud integration scenarios.

4.3.10. System for Cross-domain Identity Management

Provides: Cross-Domain Identity Management.

The System for Cross-domain Identity Management (SCIM) specifications are standardized [GWMH15] by the IETF and designed to make federated identity management easier. SCIM can be deployed as an approach to exchange and synchronize identity information required for authentication or authorization purpose. It defines a schema for representing users and groups and a protocol providing operations on these resources. The protocol works via HTTP using JSON for object representation.

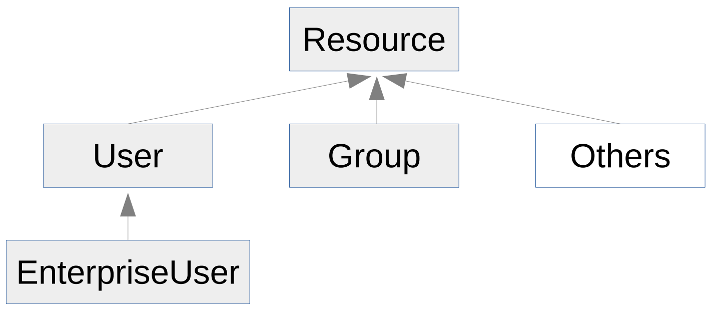


Figure 4.2.: Object Model of SCIM

The object model, pictured in Figure 4.2, is quite simple. All objects are derived from the *Resource* object. A Resource contains attributes for identification and meta information, e.g. creation date or resource type. A *User* is derived from Resource, containing user information. And further *EnterpriseUser* is derived from User, extending the object with enterprise related user information, e.g. manager or job position. *Groups* are used to model organisational structures and can contain users and/or other groups.

The objects can be retrieved, searched or updated with a REST API using HTTP `POST`, `GET`, `PUT`, `DELETE` and `PATCH`. The API defines the following operations: Create, Read, Replace, Delete, Update, Search and Bulk. Furthermore it defines endpoints to discover supported features.

4.3.11. eXtensible Access Control Markup Language

Provides: *Authorization*

The eXtensible Access Control Markup Language (XACML) is a standard from OASIS, currently in version 3.0 [OAS13]. XACML is a general purpose policy system [LPL$^+$03]. It defines a XML based syntax for a policy language and how to process those policies. The main purpose of XACML is to provide interoperability between access control implementations of different applications. It is basically an ABAC system but there also exists a profile of XACML for RBAC [OAS14].

The specification defines the work-flow of XACML in the following steps:

1. *Policy administration points (PAP)* create policies and policy sets. A policy set can contain policies, other policy sets, and policy-combining algorithms. The PAPs make the policies and policy sets available to a *policy enforcements point (PEP).*
2. A client, called access requester, sends an access request to the PEP.
3. The PEP sends the request in its native form to an *context handler*, a system which converts requests to XACML form.
4. The context handler constructs an XACML request context and sends it to the *policy decision point (PDP).*
5. The PDP requests any additional attributes from the context handler.

6. The context handler requests the attributes from a *policy information point PIP*, a system which acts as source of attribute values.
7. The PIP obtains the requested values.
8. The PIP returns the requested attributes to the context handler.
9. Optionally, the context handler adds a resource to the context.
10. The context handler sends the attributes and the resource to the PDP, which evaluates the policy.
11. The PDP returns the authorization decision, embedded in the context, to the context handler.
12. The context handler converts the context to the response format of the PEP and returns the response to the PEP.
13. The PEP permits or denies access, dependent on the received response.

The described work-flow of the comprehensive specification provide a good way to integrate decentralized policy systems. Therefore XACML is an excellent choice for distributed authorization systems [LPL+03]. But in fact this comprehensive language comes at cost of complexity and verbosity.

4.4. Software Solutions for Lifecycle Tool Integration

So far, we introduced concepts and mechanisms to fulfil the defined security requirements. The remainder of this section conceives a library and a software platform for LTIs. The section starts by introducing a community library for OSLC, the Eclipse Lyo library, and further the commercial software platform IBM Jazz.

4.4.1. Eclipse Lyo

Eclipse Lyo [ecl16] is a community project, started in 2011, with the goal to enable tool integration with OSLC. It provides a library, reference implementations, and test suites for OSLC and TRS.

Library `OSLC4J` is a Java Software Development Kit (SDK) which helps to implement OSLC and TRS tools easily. The focus of the community is in Java; however, JavaScript and further languages are planned.

The SDK provides functionality to implement clients and servers conforming to OSLC specifications, including methods for authorization of messages by HTTP Basic, HTTP Form, and OAuth 1.0a. The library was deployed for the implementation of our prototype, described in Chapter 6.

Reference Implementation The contained reference implementations demonstrate the use of the Eclipse Lyo library for various OSLC domain specifications as well as sample implementations of TRS.

An example OSLC service provider, connected to a Bugzilla[4] bug-tracker database, is used in the threat and gap analysis of the next chapter. Further, a reference implementation of a TRS service provider, in the requirements management domain, was utilized as the base for the TRS implementation of our prototype, described in Chapter 6.

4.4.2. IBM Jazz

The IBM Jazz [Cornd] platform is a set of commercial software tools for lifecycle management. The provided functionality, including OSLC interfaces, a TRS client and a SPARQL database make it to a suitable reference for the Platform Scenario.

IBM Rational Collaborative Lifecycle Management delivers solutions for requirement management, quality management, change and configuration management, as well as project planning and tracking. It is compromised by the products Rational Team Concert, Rational Quality Manager, and Rational DOORS Next Generation.

The IBM Jazz solution is analyzed in the context of the defined security requirements in the gap analysis of the next chapter.

Architecture The base component is the Jazz Team Server, a Java based web application running in an IBM WebSphere Application server. All into the platform integrated tools are registered at the Jazz Team Server. After the registration, the tools can communicate with each other. The team server acts as a central point for lifecycle project management and user administration. The team server provides a central user database that is shared with all connected

[4]https://www.bugzilla.org (Accessed: 2017-01-08)

tools. Furthermore, it allows synchronization with a Lightweight Directory Access Protocol (LDAP).

For integration of external tools into the Jazz web container, a software development kit (SDK) is provided. A more loosely integration is enabled by integrating the tools via OSLC. For this purpose, the IBM Rational Team Concert application provides an OSLC adapter.

Authentication The Jazz Team Server provides different methods for authentication: Java EE container authentication for the IBM core tools, and redirection of the authentication to the Jazz Team Server for other tools. Container authentication allows SSO based on HTTP Form, Basic authentication or with client certificates. Further, authentication via OpenID Connect and the Kerberos protocol are an option. Kerberos can only be used for tools deployed in a WebShere Application Server using Microsoft's Windows Active Directory.

For tool to tool authentication, OAuth 1.0a and OpenID Connect can be deployed. Especially tools integrated with OSLC use those mechanisms to grant authorisation for requests. Each pair of tools need to be registered as "friends," and the tools need to store a secret to secure this communication. OpenID Connect requires no pair-wise authorization; the user is authenticated via a Jazz Authorisation Server and SSO is supported.

Authorisation The Jazz platform uses a role-based authorization system. However, the authorization system only works for tools integrated into the web container using container authentication. Other tools are themselves responsible for access control. Tool to tool authorization is done via OAuth 1.0a or OpenID Connect, as described above.

IBM Lifecycle Query Engine (LQE) The LQE can be used to extract resources from lifecycle tools via TRS, index them and make the resources available for the other tools. Connected tools and users can search resources in a central database, for instance with SPARQL queries.

4.5. Conclusion

In this chapter, concepts, mechanisms, and software solutions in the context of securing lifecycle tool integrations were introduced. The chapter starts by describing levels of trust which will be further used for defining trust relationships between integrated tools. Next, common models for access control were introduced, since understanding of those is favorably for authorization concepts applied in this work. Section 4.3 lists security mechanisms which are analyzed in the subsequent threat analysis. As an outcome of this analysis, some methods were implemented with a prototype for evaluation purposes. Concluding, a library for OSLC, and an existing commercial solution for LTI were described.

The intention of this chapter is not to make a technology decision. Therefore, Chapter 5 provides a threat and gap analysis concluding on which of the described methods of Section 4.3 will be implemented with a prototype.

5. Threat and Gap Analysis

For the decision on an implementation of a prototype a threat and gap analysis was done. The analysis starts with defining the methodology and assets, deriving threats for the scenarios, and concluding security objectives and countermeasures. Next, in a threat analysis, the security objectives are discussed. The gap analysis discusses which and how existing solutions cover the defined threats and security objectives. Finally, concluding from the analysis, the decision making of the methods used for the prototype is explained.

The focus of the threat and gap analysis lies on securing tool communication (OSLC and TRS), authorization of access to resources, confidentiality of user credentials, and access to the triple store (e.g. via SPARQL interface). Security within lifecycle tools, as well as security of user data and resources at the integration platform are considered as out of scope, as they are executed within the tools as part of proprietary processes, and therefore are excluded from inter-tool data flows.

5.1. Definitions and Methodology

5.1.1. Methodology

The analysis model uses concepts from a model proposed by Zefferer and Zwattendorfer [ZZ14]. Their model for the evaluation of server-based signature solutions is based on concepts of the Common Criteria [com13]. The authors of the model highlight the possibility to use it with different implementations. *"The proposed evaluation model is based on an abstract architectural model for server-based signature solutions and can hence be applied to arbitrary implementations."* [ZZ14] Therefore, it suits well to our needs of evaluating an abstract scenario of an implementation in the domain of LTI. Subject to the different domain, we adjusted the model to the needs of developing a prototype of an LTI implementation. Figure 5.1 gives an overview of the adjusted model.

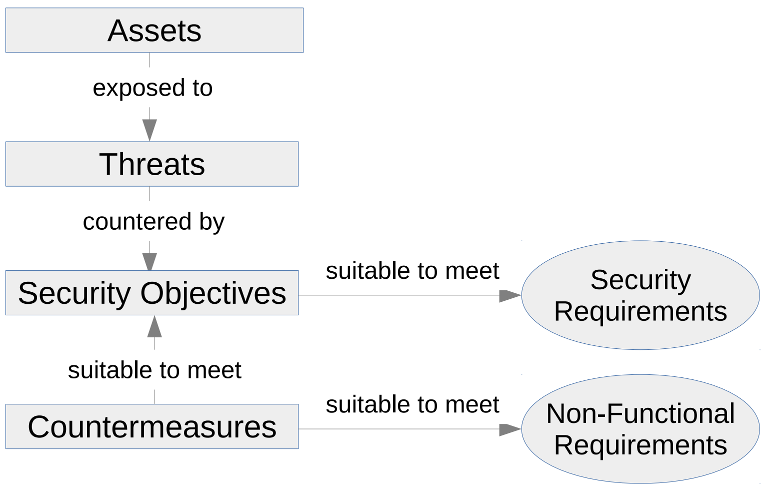


Figure 5.1.: Methodology of the Threat Analysis

A definition of each entity in the model is given in the subsequent sections. The workflow of the evaluation consists of the the following steps:

1. Assets, which need to be protected, are defined.
2. The threats, assets are exposed to, are derived.
3. Security objectives which counter the threats and meet the security requirements are deduced.
4. Countermeasures, suitable to meet the security objectives and additionally the defined non-functional requirements, are discovered and discussed.

5.1.2. Assets of the OSLC Scenario

"Assets are values that need to be protected" [ZZ14]. According to ISO/IEC 27000 *"there are many types of assets, including (a) information; (b) software, such as a computer program; (c) physical, such as a computer; (d) services; (e) people, and their qualifications, skills, and experience; and (f) intangibles, such as reputation and images."* [ISO09]

With the notation $A.o.x$ we define assets of the OSLC scenario, whereas x consecutively numbers the assets. The defined assets are scoped to the scenario of an OSLC tool interaction. Values within a tool are defined out of scope. Further, the threats are analyzed on an abstract level, omitting assets and threats to specific authorization or authentication methods. Specific security considerations to the implemented methods of the prototype are discussed in Chapter 7.

A.o.1 Communication channels between OSLC services. Confidential data including OSLC resources and authentication credentials might be exchanged between lifecycle tools. The entire data in transit is defined as an asset.
A.o.2 Retrieval of resources managed by an OSLC service. The client shall be able to rely on the OSLC service for retrieving resources from a service provider in a confidential and trustworthy way. The service for retrieval does not allow to alter resources in any way. But resources may be altered in a second step by another service, *(A.o.3-A.o.5)*.
A.o.3 Creation of resources managed by an OSLC service. The client shall be able to rely on the OSLC service for creating resources at a service provider in a confidential and trustworthy way.
A.o.4 Update of resources managed by an OSLC service. The client shall be able to rely on the OSLC service for updating resources at a service provider in a confidential and trustworthy way.
A.o.5 Deletion of resources managed by an OSLC service. The client shall be able to rely on the OSLC service for deleting resources at a service provider in a confidential and trustworthy way.
A.o.6 Credentials for user authentication, including passwords, biometrics, and private key material of the user. Secret credentials of an user for access to a service provider shall be kept confidential from unauthorized parties as well as from client tools.
A.o.7 OSLC resources hold by lifecycle tools. OSLC resources shall be kept confidential from unauthorized parties.

5.1.3. Assets of Platform Scenario

In the Platform scenario, OSLC is used to query and alter resources. Therefore all assets of the OSLC scenario need to be protected. Further, assets which are defined in this section, expand the introduced assets for the OSLC scenario.

A.p.1 Retrieval of TRSs. The client shall be able to rely on the service for retrieving change logs of OSLC resources, called Tracked Resource Sets (TRSs), from a service provider in a confidential and trustworthy way. The service for retrieval does not allow to alter resources in any way. But resources may be altered in a second step by another service, *(A.o.3-A.o.5)*.

A.p.2 Access to resources of the triplestore via SPARQL queries. The service does not allow to alter resources in any way. The service shall be protected from unauthorized access.

5.1.4. Threats

Threats potentially compromise the assets' security [ZZ14]. For the analysis, we identified general types of threats which may harm an implementation of the OSLC or Platform scenario, focused to the previously defined assets.

Potential threats were observed by research from different sources. The *STRIDE* threat model [HLOS06] defines categories of threats which were matched with our scenarios, the *Guidelines for Writing RFC Texts on Security Considerations* [RK03] and the OWASP Top 10 [OWA16] renders more precisely threats for the Internet environment. Bhatti, Bertino and Ghafoor [BT11] published in their book threats specific to identity management system like OAuth and SAML.

In the following paragraphs we describe and discuss the observed threats.

T.1 Disclosure of Communication. Eavesdropping or Man-in-the-Middle (MITM) attacks may be used to read data from the communication between tools, or the integration platform. The eavesdropper may obtain all communication data, parts of it, or may only be able to recognise patterns in the communication data.

T.2 Message Insertion/Update/Deletion. An attacker is able to insert, update, or delete messages from the communication channel. These kinds of attacks most likely may be MITM or replay attacks.

T.3 Impersonation of Identity. An attacker obtains all necessary information which is required to be authenticated as another identity. Spoofing, phishing, social engineering, or access to an authorization server can be used to obtain the information. The attacker can authenticate to tools and gains access rights in the context of the impersonated identity. Depending on the authentication method, a leaked password or token is sufficient.

T.4 Disclosure of sensible user authentication credentials. The threat of disclosed credentials in the communication is covered by *T.1*. But even providing authentication credentials of an user to other tools in the lifecycle, threats the confidentiality of this information. Other tools in the lifecycle may not be able to protect the received information in the same way the client tool can.

T.5 Unauthorized Access. Resources hold by tools, or resources of the RDF triple store in the Platform scenario are accessed by unauthorized parties.

T.6 Misuse of APIs, respective the OSLC interface, TRS interface, SPARQL interface, and APIs of the integration platform for authentication or authorization. Users may be permitted to access those interfaces, but use the interfaces in an inappropriate way with the intention to gain access without authorization.

T.7 Redirection to malicious service providers. An attacker attempts to redirect the client tool to access a malicious page instead of a real service provider.

Table 5.1.: Assets targeted by threats

	T.1	T.2	T.3	T.4	T.5	T.6	T.7
A.o.1	x	x			x		x
A.o.2	x	x	x		x	x	
A.o.3	x	x	x		x	x	x
A.o.4	x	x	x		x	x	
A.o.5	x	x	x		x	x	
A.o.6	x1)			x			x
A.o.7	x	x	x	x	x	x	
A.p.1	x	x	x		x	x	
A.p.2	x	x	x		x	x	

1) Depending on the authentication method, credentials may be transferred in plaintext.

Discussion Table 5.1 illustrates which of the defined assets are targeted by the introduced threats. The mapping between assets and threats is explained in the following discussion. A detailed discussion of assets, threats and the resulting security objectives and countermeasures is given in Section 5.2.

T.1 Disclosure of the communication threatens all messages in the OSLC communication workflow. The workflow includes retrieval, creation, update, and deletion of resources. The attacker is able to read information about the transferred resources and monitor operations. Depending on the authentication method, user credentials can be read from the communication. Furthermore, in the Platform scenario, TRS and SPARQL queries can be monitored to gain information about resources, processes, and users.

T.2 Unrecognized insertion, update, or deletion may allow access to protected resources and APIs, allowing to perform a query and other operations on the OSLC, TRS, and SPARQL interfaces.

T.3 With impersonation of a user's identity, an attacker can create, retrieve, update, delete resources, or query a triplestore with the access rights of the impersonated identity.

T.4 Disclosure of user credentials targets to the authentication credentials the user applies to sign in to a service provider or integration platform services.

T.5 An attacker can gain access to protected resources of an OSLC service provider or a triplestore by monitoring the communication or accessing the services with impersonated identity or misuse of APIs. Therefore, this threat is based on all other introduced threats.

T.6 Misusing APIs of the service provider or integration platform, no matter if done by intention or not, may allow to perform some operations without authorization or read protected information.

T.7 Redirection to malicious service providers. If the malicious service provider can convince the client to be a trustful service provider, the malicious service can record confidential data, such as OSLC resources or client credentials.

5.1.5. Security Objectives

Security objectives are derived from the defined security requirements, threats, and assets. The objectives must be able to counter all potential threats and should meet the defined security requirements from Chapter 3.

In the remaining of this subsection we introduce the derived security objectives.

For a mutual communication between the two parties both, the server and the client/user, needs to be authenticated. The need for authentication was already elaborated in Section 3.

O.1 Authentication of the server. A client initiating a communication to a server shall be able to verify the authenticity of the server. Without authentication, malicious servers can persuade the client to disclose secret information.

O.2 Authentication of the user. The server needs to authenticate the user. Authorization to resources managed by OSLC services is based on the user's identity provided.

O.3 Integrity of the OSLC, TRS, and SPARQL communication. If a party in the communication cannot check the integrity of messages, it has no assurance that the message was not modified in transit. The integrity of resources which are stored in a tool is not examined in this work, as mentioned earlier.

O.4 Confidentiality of the OSLC, TRS, and SPARQL communication. In addition to the integrity of data in transit, the information must be kept confidential. Confidentiality is not treated for resources retraining at the tools.

O.5 Grant authorized users permission to create, retrieve, update, and delete OSLC resources.

O.6 Confidentiality of the user's authentication credentials. The credentials, required to authenticate to a service provider, have to be kept secret from unauthorized parties. This includes 'trusted' parties, as the client application, other lifecycle tools, and the integration platform. Only the user and the service provider should be aware of the credentials to minimize the risk of leaked credentials.

O.7 Grant authorized client tools permission to query TRSs from a service provider.

O.8 Grant authorized users permission to query resources from a triplestore via a SPARQL interface.

5.1.6. Countermeasures

Finally, the countermeasures define a set of security methods, which have to be considered for usage in the prototype implementation. The methods were introduced in Chapter 4. The chosen methods are discussed regarding compliance of the non-functional requirements defined in Section 3.3. Table 5.2 shows which threats are addressed by the introduced countermeasures.

The following countermeasures, starting with *C.o*, were identified for the OSLC scenario.

C.o.1 Secure communication by using TLS with server certificates. HTTPS, which uses TLS, is used instead of HTTP for the communication channel between the OSLC interfaces. With server certificates, the OSLC client can verify the trustworthiness of the service provider. The service provider cannot verify the trustworthiness of the OSLC client. Therefore, the client additionally has to authenticate with some other method to the service provider.
C.o.2 HTTP basic authentication. The user is authenticated with HTTP basic authentication to the service provider. This method mandates the usage of HTTPS to protect the authentication credentials, which are transferred in plaintext. HTTP basic authentication is one of the suggested methods for resource authorization by the OSLC core specification [osl13].
C.o.3 Form based authentication. Another method which is suggested by the OSLC core specification. The OSLC client is redirected to a login form of the service provider. The user credentials are transferred in plaintext from the client to the login form; therefore HTTPS needs to be used to protect integrity and confidentiality of the credentials.
C.o.4 Authorization with OAuth 1.0a. OSLC resources are protected with OAuth 1.0a, also suggested by the OSLC core specification. At least if the `PLAINTEXT` signature method is used for the OAuth messages, HTTPS is required to protect the transferred tokens.
C.o.5 Authorization with OAuth 2.0. OSLC resources are protected with OAuth 2.0. HTTPS is required to protect the transferred tokens.
C.o.6 Authentication with OpenID Connect. The protocol is based on OAuth 2.0 and adds the features of authentication and the SSO. OpenID Connect is suggested by the OSLC core specification draft version 3.0. With OpenID Connect the tools retrieve the authentication from an identity provider, without the need to manage users authentication credentials on their own.
C.o.7 Authentication/authorization with SAML. SAML can be used similar to OAuth 2.0 and OpenID Connect for identity management, providing authorization, and authentication with SSO.

The following countermeasures, starting with *C.p*, were additionally identified for the Platform scenario. The above defined countermeasures *C.o* are also used for the Platform scenario. All of the following countermeasures have the purpose of access control and can detect misuse of APIs if a user tries to access resources without permission.

C.p.1 Access control with a centralized model. Access control to resources of the integration platform is provided by a centralized access control model, independent of access control policies of the lifecycle tools. Access policies can be assigned to resources at the integration platform's administration page.

C.p.2 Access control delegated to tools. The integration platform delegates authorization to the lifecycle tool which is in possession of the corresponding resource. The integration platform forwards the user authorization for the resource to the appropriate tool.

C.p.3 Access control with XACML. Every lifecycle tool supports the XACML protocol. The authorization decision of the integration platform is based on homogeneous policies of the lifecycle tools. Protocols like OpenID Connect or SAML might be used for identity management, required for making the authorization decision.

Table 5.2.: Threats addressed by countermeasures

	C.o.1	C.o.2	C.o.3	C.o.4	C.o.5	C.o.6	C.o.7	C.p.1	C.p.2	C.p.3
T.1	x									
T.2	x									
T.3	x			x	x	x	x			
T.4	x		x1)	x	x	x	x			
T.5	x	x	x	x	x	x	x			
T.6		x	x	x	x	x	x	x	x	x
T.7	x									

1) Only if the user credentials are not visible to the client tool. E.g. by entering the credentials via an HTML iframe hosted by the service provider.

5.2. Threat Analysis

The following analysis helps to identify the most convenient countermeasures used for the proposed solution in the next chapter. It is structured by the security objectives which need to be fulfilled. Table 5.3 shows which countermeasures cover the security objectives.

Table 5.3.: Security objectives covered by countermeasures

	C.o.1	C.o.2	C.o.3	C.o.4	C.o.5	C.o.6	C.o.7	C.p.1	C.p.2	C.p.3
O.1	x									
O.2	x1)	x	x	x	x	x	x			
O.3	∼	∼	∼	∼	∼	∼	∼			
O.4	∼	∼	∼	∼	∼	∼	∼			
O.5		x	x	x	x	x	x			
O.6	∼	∼	∼	∼	∼	∼	∼			
O.7		x	x	x	x	x	x			
O.8								x2)	x2)	x2)

x covered, ∼ partly covered

1) Only if mutual TLS with client certificates is used.

2) Assuming a previous user authentication.

Authentication - O.1, O.2 TLS with server certificates can be used to authenticate the server. The client verifies a certificate of the service provider or integration platform. To further verify the authenticity of the client, mutual TLS can be used. In a mutual TLS scenario, both communication partners have certificates which allow a mutual authentication.

However, client certificates are often not available, especially if multiple devices, including mobile devices, are used. Therefore, another method is required to authenticate the user. Different authentication methods can be deployed, including HTTP Basic, Form authentication, OAuth, OpenID Connect, and SAML *(C.o.2 - C.o.6)*.

A possible threat to the authenticity of the client is the impersonation of identities. In this case the attacker obtains authentication information of the user. The attacker may steal the information with methods as spoofing or phishing, whereas the user is redirected to a malicious site for entering username and password. Other methods are social engineering or brute force attacks to exploit insecure user passwords. Some of the attacks are only applicable if passwords are used for user authentication/authorization.

Authenticity of the server (service provider or integration platform) basically is protected by verifying the server certificate. But typically applications allow the user to accept unknown or expired server certificates. This enables attackers manifold possible attacks, including MITM, and to masquerade as a server to the client.

Summarized, client authentication methods can be used in combination with TLS to establish authenticity. The same authentication methods can be used for all connections, including OSLC, TRS, and SPARQL services.

Integrity and Confidentiality - O.3, O.4 Integrity and confidentiality of the communication channel can be protected with TLS, assuming authenticity of server and client as discussed in the last paragraph.

OSLC/TRS Operations - O.5, O.7 All OSLC/TRS operations can be executed with the same HTTP REST API. The service provider grants permission to resources based on user rights. Therefore, an authorization/authentication method can be used to authorize access to resources on behalf of the user. The same method which is used to verify the authenticity of the client, as discussed above, can be utilized *(C.o.2 - C.o.6)*. Furthermore, consumer and service provider tools need to authenticate each other, eavesdropping, MITM, or spoofing attacks cannot be prevented otherwise.

Additionally, insecure implementation of the OSLC interface or the authorization/authentication method can lead to attacks by misusing the API. Each service provider, implementing the OSLC specification, needs to take care of a correct implementation. In addition, the specifications of the utilized security methods have to be implemented carefully, kept up-to-date, and comply with possible security considerations of the specifications. Protocols as OAuth 2.0, SAML, or XACML are comprehensive and require a large effort to implement them carefully. The usage of libraries can reduce the work. We will further discuss the issue of the implementation effort in Chapter 7.

Confidentiality of user credentials - O.6 Although confidentiality of the communication can be assured by TLS, authentication credentials, in addition need to be protected against the other tools in the lifecycle. Some authentication methods, as HTTP Base or Form authentication, require transmitting authentication credentials from the client to the server.

Confidentiality of authentication data can be provided by token, assertion, or redirection based methods. Some implementations of HTTP Form authentication, as well as OAuth 1.0a, or OAuth 2.0 redirect the users to enter their password in a form, hosted by the service provider, unseizable for the client application. Further, SSO methods as OpenID Connect or SAML can be used, whereas only the authorization server needs to know the users' credentials. OAuth and OpenID Connect use tokens for the communication instead of passwords, SAML uses assertions. In Chapter 6 the detailed data flow, including authentication data, will be illustrated for HTTP Basic authentication, OAuth 1.0a, and OAuth 2.0. Further in Chapter 7, we discuss security issues related to confidentiality of user credentials with those methods.

Possible attacks are spoofing, phishing or social engineering. Those threats are covered by methods protecting the confidentiality of the communication and authenticity of the communication partners.

To improve the protection of credentials, client authentication methods, whereas the client application does not need to be aware of the user's credentials, such as OAuth, OpenID Connect, or SAML, should be used.

SPARQL Query - O.8 For the protection of the SPARQL communication and the authentication, the same holds as discussed for OSLC operations. But, the authorization to resources in the triple store is an additional difficulty. The integration platform needs to make authorization decisions, which users are allowed to access which resources of the triple store.

The used access control system may induce additional threats. If access control is delegated to tools *(C.p.2)*, the extra communication between the tool and the integration platform needs to be protected. In the scenario of using XACML *(C.p.3)* the required components, the policy decision and policy enforcement points need to be protected. Making the access control decisions directly via the integration platform *(C.p.1)*, only requires protecting the query API of the platform, which is basically the same problem as the protection of the OSLC communication.

Usability, Configuration, Interoperability. Beside threat resistance, usability, simple configuration and administration, a well as interoperability and lightweight integration are important criteria to choose suitable countermeasures. These non-functional requirements were introduced in Section 3.3 and are discussed regarding security objectives and countermeasures in this paragraph.

TLS is easy to install and use with HTTPS, but it requires to install server certificates which are accepted by all clients. Mutual TLS further increases the security by client authentication, but requires a client certificate infrastructure.

SSO and no SSO can categorize the client authentication/authorization mechanisms. HTTP Basic and Form authentication, and OAuth authorization provide no SSO; hence the user has to login to every service provider. OpenID Connect and SAML can be configured as SSO provider to increase the usability of the system.

The intention of SSO is to remove redundancy in administration of multiple logins. But in terms of configuring the system, the SSO solutions may require installing additional identity providers. Further, synchronization and transfer of identities increases the effort of configuration and maintenance. Although, some corporations may already have implemented an identity management system, which can be used to provide identity information for SSO.

Form authentication and SAML have further issues with interoperability. Form authentication is not standardized, therefore service providers have to define the workflows leading to diverse flows. The clients have to implement multiple authentication workflows for different service providers. Also SAML has issues with its SOAP based communication on different client types, [NB13] describe this issue and mentions the usage of OAuth related protocols instead of SAML. In contrast, OAuth 2.0 uses HTTP REST, similar to the OSLC protocol, and defines multiple device flows, including mobile devices.

Regarding the authorization method for the integration platform, the introduced countermeasures have different advantages and disadvantages. A centralized access control model *(C.p.1)* is easy to understand by the users, needs no custom implementations by the service providers, and therefore is adequate in terms of interoperability. On the other hand, each service provider has its own access control system in addition to the integration platform, inferring redundancy and extra work for the system administrator. Another way is to delegate access control to the tools *(C.p.2)*. Querying resources from the service provider via OSLC can be used to check if the user is allowed to access a resource. Hence, the workload of the system administrator is reduced and interoperability is still given. But this method destroys some advantages of the centralized data store in the integration platform. The delegated access control check requires licenses for the tools, needs the tools are always online, and additionally decreases the performance dramatically. To get the advantages of the centralized method without the problems of the tool delegation method, XACML *(C.p.3)* can be deployed. The lifecycle tools provide access control policies to the integration platform and the platform can enforce the policies without the need to contact the tool for every access request. However, such an XACML enforced system requires a lot of additional configuration work and every tool needs to implement the XACML workflow. Different lifecycle tools have very diverse access control systems, the effort for adoption to XACML might not be reasonable in many cases.

5.3. Gap Analysis

After defining and discussing threats and countermeasures, we discuss threats which are not covered by existing solutions for LTI in a gap analysis. The gap analysis, combined with the previous threat analysis, helps to identify suitable security methods for implementing a prototype. Afterward, the prototype will be evaluated in Chapter 7.

The gap analysis is done for existing solutions which already were introduced as related work in Section 4.4. Implementations of the OSLC core specification were analyzed with two different authorization methods, HTTP Basic authentication and OAuth 1.0a. An example from the Eclipse Lyo project [ecl16], connecting to a Bugzilla [bug16] service provider, was used therefore. Further, the IBM Jazz platform [Cornd], in major version 6, was tested by using Basic authentication, OAuth 1.0a, and a Form authentication method.

5.3.1. Discussion of Threats

The different solutions are targeted by the same threats, depending on the authentication method. An overview, which threats are counteracted by the solutions, is given by Table 5.4. For the Jazz platform no open source code is available, hence some countermeasures may be implemented which we do not know about.

The following paragraphs are structured according to the applied user authorization/authentication method. HTTPS with server certificates is assumed for all scenarios, consequently, the clients can verify the authenticity of the service provider. Thus, confidentiality and integrity of the communication channel can be assured by verifying the authenticity of the user by the server.

Basic Authentication HTTP Basic authentication is a plain form of user authentication which can be used for LTI. Both, the Eclipse Lyo libraries and IBM Jazz, provide an implementation of this authentication method.

Sending the username and password in plaintext implies some major threats, even if a secure communication line with TLS is established. The specification of HTTP Basic authentication, RFC 2167 [FHBH$^+$99a], already lists potential threats which are discussed in the following for the LTI scenario. Secure storage of the passwords on the server is supposed as a duty of the service provider and not discussed here.

One major problem is the user in person. Using weak passwords exposes them to brute force attacks. The server can counterfeit, by setting password policies, impacting the usability of the system with long and complex passwords.

Further, a user might accept invalid certificates of the server, making spoofing or MITM attacks possible. The server can act as a proxy and read user credentials from the authorization header, or masquerade as a service provider and receive requests from the client including the authorization information [OHB06].

At last, the client application type may be a problem. HTTP Basic authentication is developed for web applications. The browser asks for username and password and adds the authorization header to the HTTP request. But for native workstations or mobile applications typically the authentication is handled by the application itself. Thus, it appears that the client application has access to the username and password at the service provider in plaintext. A special trust relationship between server and client is required, which is not given for many scenarios.

HTTP Digest authentication [FHBH$^+$99a] can be used to counterfeit some of the described attacks, including brute force, MITM, and spoofing attacks. But modifications of those attacks, such as dictionary, or precomputed dictionary attacks in combination with MITM or spoofing still threaten the authentication.

More issues and threats are discussed at Chapter 7, when evaluating the prototype implementation. In conclusion, HTTP Basic is the most simple and a widely accepted authentication method, but is *"[...] very much on the weak end of the security strength spectrum."* [FHBH$^+$99a]

OAuth 1.0a and OAuth 2.0 Authorization OAuth can be implemented for delegated access control of resources, where the authorization is granted by the user. OAuth 1.0a implies a range of security considerations [OAu09], the same holds for OAuth 2.0 [LMH13].

The OAuth 1.0a protocol mandates to use either `PLAINTEXT`, `HMAC-SHA1`, or `RSA-SHA1` to sign OAuth messages. `PLAINTEXT` fully relies on a secure communication channel. The other signature methods further protect integrity and authenticity of the messages as long as the consumer secret is kept secret. OAuth 2.0 always mandates the use of TLS.

Spoofing and MITM attacks can be applied for eavesdropping of the message content. Those, exchanged security tokens may be stolen and misused. Service providers can minimize the risk by limiting the scope and validity of tokens. Security tokens should be cryptographic secure in a way that it is practically impossible to expose the token within its validity time range by usage of brute force or similar attacks. Further, service providers should not solely rely the authenticity of the client on the OAuth consumer secret, additional information, such as the IP address of the client, should be verified.

The user has to struggle with phishing, spoofing, MITM, and cross-site request forgery (CSRF) attacks. With incorporated service providers, spoofing and MITM attacks enable attackers to reveal the consumer secret or tokens. Moreover, the user is redirected to a login page of the service provider. This procedure endangers to phishing, CSRF, and clickjacking attacks for catching the user's password.

In conclusion, OAuth is a suitable solution to protect the users credentials against the client application. However, when implementing the OAuth protocol both, service provider and consumer, have to consider many potential vulnerabilities. The evaluation, in Chapter 7, discusses further issues.

Form and SSO Authentication The OSLC specification [osl13] lists HTTP Form authentication as a possible way for resource authorization based on user authentication. The IBM Jazz platform uses this method per default. Typically a request to a service provider is redirected, the user enters username and password, and finally the client can send requests authenticated by an HTTP session cookie. A workaround for redirection is to send the username and password directly to a Java servlet of the Jazz service provider. Form authentication is non-standardized and therefore further will not be discussed.

IBM Jazz implements OpenID Connect to allow an OAuth-like workflow with an SSO feature. No available open-source solution, including the Eclipse Lyo project, could be found to include OpenID Connect in this analysis.

Table 5.4.: Threats addressed by existing solutions

	OSLC with Basic Auth 1)	OSLC with OAuth 1)	IBM Jazz 2)
T.1	x	x	x
T.2	x	x	x
T.3	x	x	x
T.4		x	x
T.5	x	x	x
T.6			– 2)
T.7	x	x	x

1) Using TLS (HTTPS) with server certificates.

2) The source code of the Jazz platform is not public. Countermeasures against API misuse cannot be verified.

5.3.2. Discussion of Security Objectives

Table 5.5 displays a mapping of security objectives addressed by existing solutions. In overall it depends on the use of TLS and user authentication/authorization methods, which address authentication of the server and user, integrity and confidentiality of the communication, protection of OSLC and TRS operations, and some authentication/authorization methods protect the confidentiality of user credentials *(O.1 - O.7)*. A further issue which was found is the protection of SPARQL queries from the triple store of the integration platform *(O.8)*. This will be discussed in the following paragraph.

SPARQL Query Authorization The Jazz platform implements an RBAC model to enforce access rights, based on the user's identification or the user's mapping to a role. No reference project to read from a triplestore is implemented with the Eclipse Lyo project.

On the administration page of the Lifecycle Query Engine (LQE), access can be granted to users or groups of users (roles). Access can be granted for all resources of the triplestore or to resources of individual TRS sources. This

Table 5.5.: Security objectives addressed by existing solutions

	OSLC with Basic Auth1)	OSLC with OAuth1)	IBM Jazz 1)
O.1	x	x	x
O.2	x	x	x
O.3	x	x	x
O.4	x	x	x
O.5	x	x	x
O.6		x	x2)
O.7	x	x	x
O.8			x3)

x covered, ∼ partly covered

1) Using TLS (HTTPS) with server certificates.

2) Depending on the user authorization method.

3) Using RBAC model defined in Jazz Team Server, similar to *C.p.1*

system is easy to understand for the administrator, and the lifecycle tools do not need to implement some access control protocol, as XACML. However, it allows no fine-grained access control and adds redundancy by having multiple access control systems within the lifecycle.

The access control policies for the triplestore may diverge from the policies of the resource holding lifecycle tools. Users might get access to resources without permission of the resource holding lifecycle tool.

5.4. Conclusion

Concluding from the threat and gap analysis a selection of methods used for the prototype needs to be done. As a base principle, only standardized security methods are considered to increase interoperability. This *standardized approach* uses existing technologies to solve open problems instead of developing new solutions.

> *Standardized approach.* With so many schemes in various stages of adoption, it is only prudent for organizations to take an incremental, ”integrateable” approach, designing new solutions that complement existing standards. [BBG07]

Decisions for the Prototype Implementation. A distinct finding of the threat and gap analysis is the need for securing the communication channels. All of the used LTI solutions (OSLC, and other) are based on HTTP requests. For this reason, adoption to HTTPS, which applies TLS, will be a general assumption for the prototype implementation.

The OSLC specification [osl13] recommends HTTP Basic authentication, OAuth 1.0a, and Form authentication. The decision was made to exclude Form authentication because its diverging implementations would contradict the standardized approach. HTTP Basic and OAuth 1.0a will be used to authenticate OSLC and TRS communication with the prototype. The libraries from the Eclipse Lyo [ecl16] can be used to reduce implementation work.

Many service providers redeemed OAuth 1.0a by OAuth 2.0 in the last years, thus OAuth 2.0 will be implemented and evaluated additionally.

Out-of-Scope and Future Work. To limit the scale of the prototype and its evaluation, SSO methods and access control methods for the Platform scenario will be omitted and need to be discussed in future work.

OpenID Connect and SAML have distinguished as possible candidates for a SSO requirement. OpenID Connect may be preferred as an extension of OAuth 2.0, SAML may be preferred in a corporate environment with an existing SAML infrastructure.

For access control of e.g. an SPARQL interface in the Platform scenario, we have identified three different possible solutions. The delegation of access control to the lifecycle tools may have too many disadvantages, especially the need of always online and accessible tools and the additionally required time for access control. A centralized access control model produces redundancy in administrating policies, but is easy to understand and implement. A draft specification of the OSLC initiative, the Indexable Linked Data Provider Specification [osl15a], extends this approach by allowing to query access contexts from the service providers. This approach helps to reduce redundancy and unifies access policies. Nevertheless, a corporation may seek to harmonize policies from all tools to a common ground. XACML can be utilized for such a scenario. For veritable results, implementations of those methods need to be compared and evaluated in future work.

6. Implementation

Based on the findings of the previous chapters, a prototype was developed. The implementation illustrates the introduced scenarios in Section 2.3 with respect to the defined security requirements in Chapter 3.

In this chapter, the prototype is described in detail, including an overview of the architecture, a brief description of each component, data flows between the components, and a detailed description of the authorization methods.

6.1. Architecture

6.1.1. Overview

The prototype, shown in Figure 6.1, consists of three main parts; RM Tool, Refine, and Integration Platform; which are described in more detail in the next section. The components are coupled to each other using the requirements management specification of OSLC and TRS. The connection between RM Tool and Refine reflects a typical OSLC scenario with two tools connected via OSLC. To illustrate the Platform scenario the components Integration Platform and RM Tool were used together with an RDF triple store providing an SPARQL interface.

6.1.2. Components

All of the following components were placed at disposal, the task was to modify the components in terms of authorization and connect them together to a tool-chain. The following paragraphs describe the components and highlight the initial state, modifications, and extension.

RM Tool The RM Tool is a Java web server providing requirements [osl12] by an OSLC interface, a prototype of a requirements management provider.

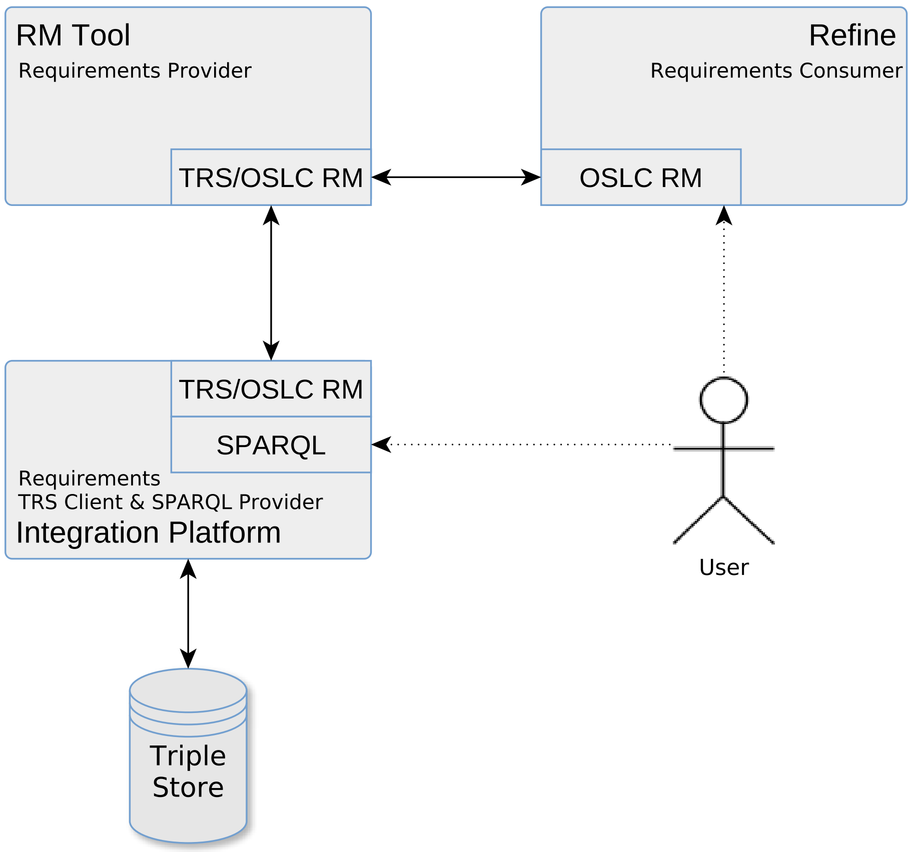


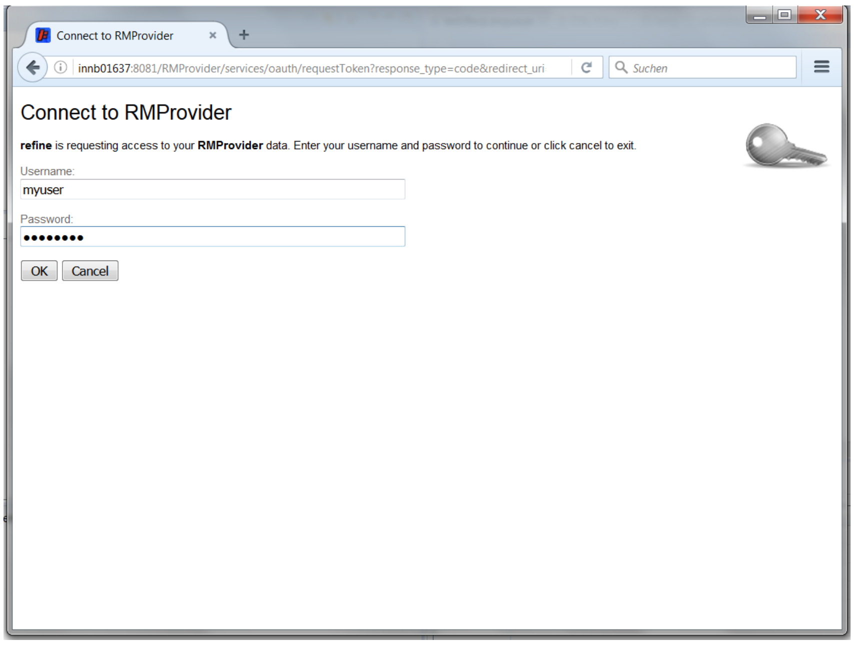
Figure 6.1.: Architectural overview of prototype implementation

The purpose of RM Tool is to store requirements, and offer an OSLC interface to other tools for CRUD operations to the resources.

Initial State: The server has provided functions to retrieve, update, and store OSLC resources. For the usage as a prototype, persistence is solved file based. Further, HTTP Base authentication was implemented for authentication of requests. For test purpose, RM Tool was connected to an IBM Jazz platform.

Modifications:

- Implementation of the TRS specification [osl15b] to provide changelogs of requirements.
- Service provider implementation of the OAuth 1.0a [OAu09] specification for authorization by an user. Figure 6.2 shows the user authorization page for OAuth.
- Service provider implementation of two-legged OAuth 1.0a for authorization by a trusted client without user interaction.
- Service provider implementation of the OAuth 2.0 [Har12] specification with grant type Authorization Code for authorization by a user.
- Service provider implementation of OAuth 2.0 with grant type Resource Owner Password Credentials for authorization by a trusted client without user interaction.


Connect to RMProvider
+
innb01637:8081/RMProvider/services/oauth/requestToken?response_type=code&redirect_uri
Suchen

Connect to RMProvider

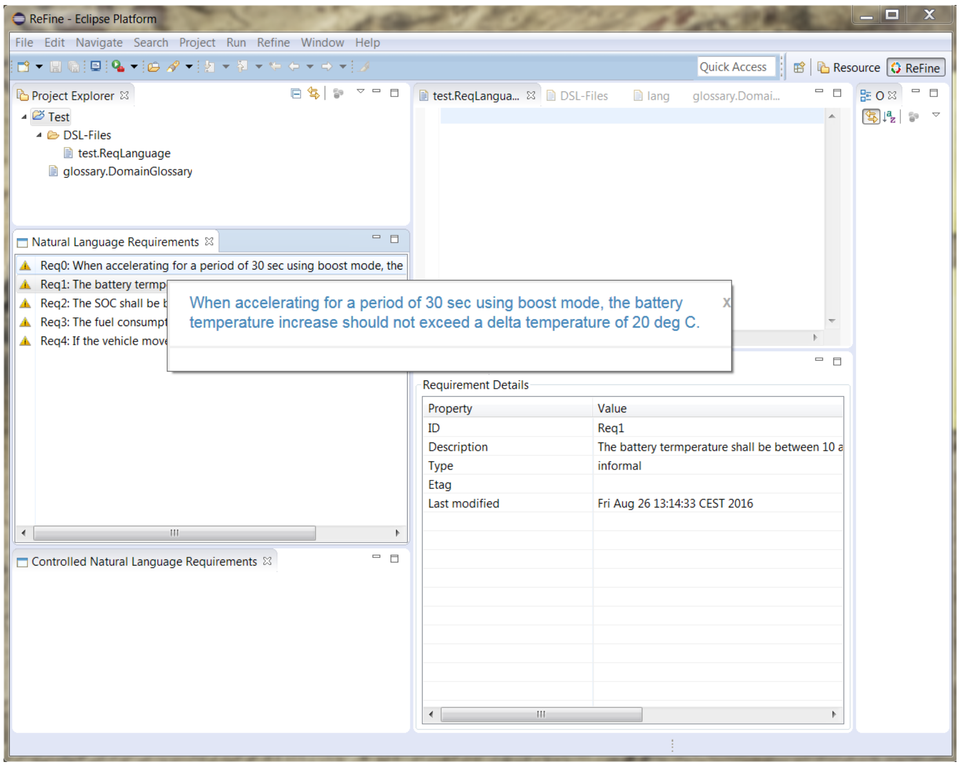
refine is requesting access to your **RMProvider** data. Enter your username and password to continue or click cancel to exit.

Username:
myuser

Password:
●●●●●●●●●

Figure 6.2.: OAuth user authorization page of the RM Tool

Refine VIRTUAL VEHICLE Refine is a tool which allows retrieving requirements using OSLC, altering (refine), and updating them at the requirement management server. A use case of Refine is described with the OSLC Scenario 2.3.1. Figure 6.3 shows a screenshot of the main view after loading OSLC requirements from RM Tool.



Req0: When accelerating for a period of 30 sec using boost mode, the
Req1: The battery temp
Req2: The SOC shall be b
Req3: The fuel consumpt
Req4: If the vehicle mov

When accelerating for a period of 30 sec using boost mode, the battery temperature increase should not exceed a delta temperature of 20 deg C.

Property	Value
ID	Req1
Description	The battery termperature shall be between 10 a
Type	informal
Etag	
Last modified	Fri Aug 26 13:14:33 CEST 2016

Figure 6.3.: Refine, displaying sample OSLC requirements from RM Tool

Initial State: Refine is implemented in Java and uses the Eclipse SWT [Fou17] toolkit. For authorization of requests to retrieve resources, load resource previews, and update the resources, Refine already provided two different methods: HTTP Basic and authentication to IBM Jazz with form authentication.

Modifications:

- Consumer implementation of OAuth 1.0a to connect to OSLC service providers, authorized by the user.
- Consumer implementation of OAuth 2.0 with grant type Authorization Code to connect to OSLC service providers, authorized by the user.

- Automatic redirection to login/authorization page of the service providers, to authorize the connection by the user.
- Web service handling OAuth 1.0a and OAuth 2.0 callbacks after the user authorized the connection at the service provider.
- Minor changes to the OSLC interface to allow connections to RM Tool.

Integration Platform The Integration Platform is a prototype implementation of a TRS client for the domain of requirements management. It is equipped to collect OSLC requirements from multiple TRS providers, and store them in an RDF triple store.

Initial state: The Integration Platform is implemented in Java and connects to an Eclipse RDF4J [Fou16b] triple store. Figure 6.4 shows a screen-shot of the rdf4j web administration page. Connections to TRS providers can be authorized with HTTP Base authentication.

Modifications:

- Support of the OSLC requirements domain.
- Implementation of OAuth 1.0a and OAuth 2.0.
- Installation and configuration of a rdf4j triple store.

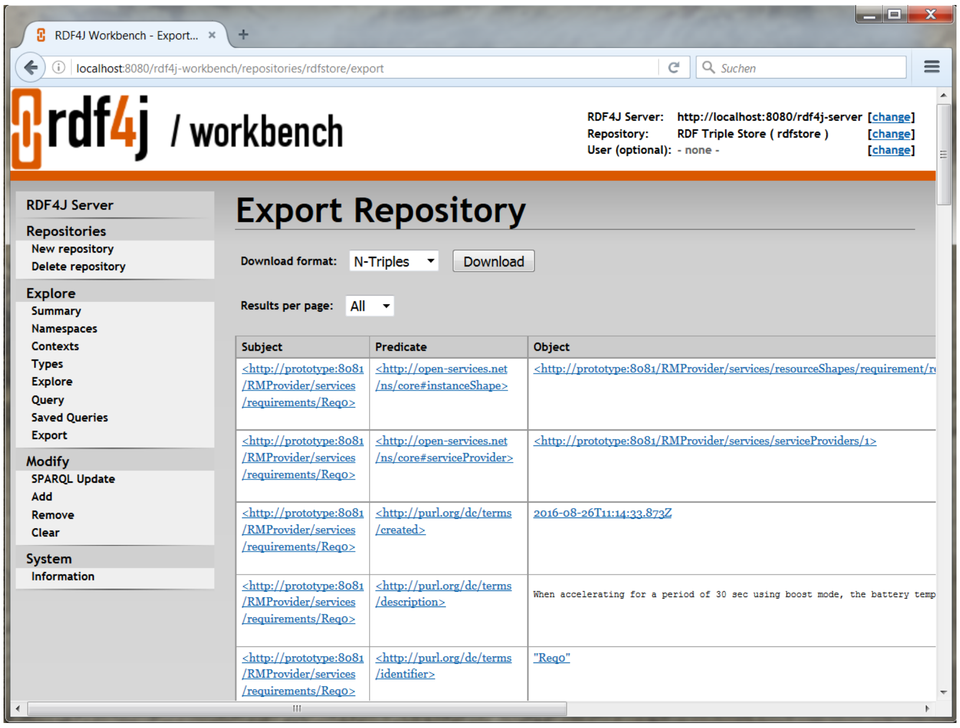
6.1.3. Libraries

Eclipse Lyo The Eclipse Lyo library [ecl16], introduced in Section 4.4.1, is used by all components to handle the OSLC and TRS communication. Furthermore, the library provides methods to authorize the communication.

Initial state: Interface for OSLC and TRS communication; authorization with HTTP Basic and OAuth 1.0a.

Modifications:

- Extended the OAuth 1.0a authorization provider web service by OAuth 2.0 grant types Authorization Code and Resource Owner Password Credentials.



RDF4J Server: http://localhost:8080/rdf4j-server [change]
Repository: RDF Triple Store (rdfstore) [change]
User (optional): - none - [change]

Export Repository

Download format: N-Triples

Results per page: All

Subject	Predicate	Object
<http://prototype:8081/RMProvider/services/requirements/Req0>	<http://open-services.net/ns/core#instanceShape>	<http://prototype:8081/RMProvider/services/resourceShapes/requirement/r
<http://prototype:8081/RMProvider/services/requirements/Req0>	<http://open-services.net/ns/core#serviceProvider>	<http://prototype:8081/RMProvider/services/serviceProviders/1>
<http://prototype:8081/RMProvider/services/requirements/Req0>	<http://purl.org/dc/terms/created>	2016-08-26T11:14:33.873Z
<http://prototype:8081/RMProvider/services/requirements/Req0>	<http://purl.org/dc/terms/description>	When accelerating for a period of 30 sec using boost mode, the battery temp
<http://prototype:8081/RMProvider/services/requirements/Req0>	<http://purl.org/dc/terms/identifier>	"Req0"

Figure 6.4.: RDF triples from RM Tool stored in rdf4j triple store of the Integration Platform

- Helper class for OSLC and TRS clients to connect to service providers via OAuth 2.0 grant types Authorization Code and Resource Owner Password Credentials.

net.oauth Initially developed by Netflix Inc., and extended by Google Inc., net.oauth [net10] is a Java library providing functionality for creation and validation of OAuth 1.0a messages, including signature creation and validation.

Usage: The library is used for consumer and service provider implementation of OAuth 1.0a. Therefore, it is referenced by the Eclipse Lyo packages and the RM Tool.

Apache Oltu Apache Oltu [Fou16a] is a Java library for OAuth 2.0, furthermore, it covers also implementations for related protocols as JWT, and OpenID Connect.

Usage: The library is used for consumer and service provider implementation of OAuth 2.0. Therefore it is referenced by the Eclipse Lyo packages and the RM Tool.

6.1.4. Sample Use-Cases

This section illustrates a sample use-case of the prototype implementation. The RM Tool represents a requirements management software; Refine a third-party tool to modify requirements, and the Integration Platform regularly loads modified requirements from RM Tool and stores them to an RDF triple store. For authorizing the access to requirements OAuth is used.

Requirement refinement. A user wants to refine existing requirements from RM Tool, using a third-party tool Refine.
First, the user starts Refine and loads requirements from RM Tool. The user has to authorize the transfer with OAuth. Therefore, the user is redirected to an authorization web page of the RM Tool. After granting authorization, the requirements are loaded via OSLC. The user can now refine some requirements. Finally, Refine links the updated resources to the original requirement and updates them at the RM Tool. The same OAuth access token can be used for updating the requirements. Repeating the user authorization step is not necessary.

Search for requirements. A user wants to search requirements from different service providers.
The Integration Platform is capable of connecting to multiple requirements provider. It uses functional users[1] of the service providers to load resources via OAuth without user interaction. The two-legged mode of OAuth 1.0a requires authorization for each message, OAuth 2.0 allows retrieving an OAuth token with the grant type Resource Owner Password Credentials. Regularly, change logs from service providers are loaded and modifications are applied to an RDF triple store. Now the user can query the database via an SPARQL interface.

[1]A functional user is utilized by a client tool to access resources of a service provider, with access rights of the functional user. The service provider maps the functional user uniquely to the appropriate client tool. Thereby, a client tool can access resources of a service provider without the need of authenticating with a normal user.

6.2. Data Flow without Authorization

This section describes the data exchanges between the components of the prototype. Authorization of the communication is further explained in the subsequent section. Messages of a sample data flow can be found in Appendix A.

6.2.1. Refine - RM Tool

Figure 6.5 shows the data flow between Refine and RM Tool for the case of loading requirements.

Step 1: At first Refine loads the rootservices[1] (Listing A.1) document from RM Tool. Access to this document requires no authorization. The document contains URLs for OAuth authorization, and service provider catalogs.

Step 2: After obtaining authorization, Refine loads the service provider catalog (Listing A.2) which contains further OSLC information including URLs for Query Capability to query resources and URLs for Creation Factories to create OSLC resources.

Step 3: A list of requirements is retrieved from the query capabilities (Listing A.3) URL.

Step 4: Each requirement from the list is loaded with an HTTP GET from RM Tool (Listing A.4) and displayed in Refine.

6.2.2. Integration Platform - RM Tool

Figure 6.6 shows the data flow between the Integration Platform and RM Tool for the case of initially loading requirements from a TRS provider. The initial flow includes loading resources from a base repository in addition of processing change logs, described in the following steps.

Step 1: After obtaining authorization, the Integration Platform loads the TRS (Listing A.5) from RM Tool by an HTTP GET. It contains a change log with the latest changes of requirements and a link to a base resource, described in step 2.

[1]The OSLC specification [osl13] defines the OAuth URLs in the service provider catalogs. The solution from IBM Jazz [Cornd] additionally requires rootservices on top of the service provider catalogs. The rootservices document includes ULRs to service provider documents as well as OAuth URLs. A sample document is listed in Appendix A. For testing purposes with the IBM Jazz platform, the prototype was implemented with rootservices.

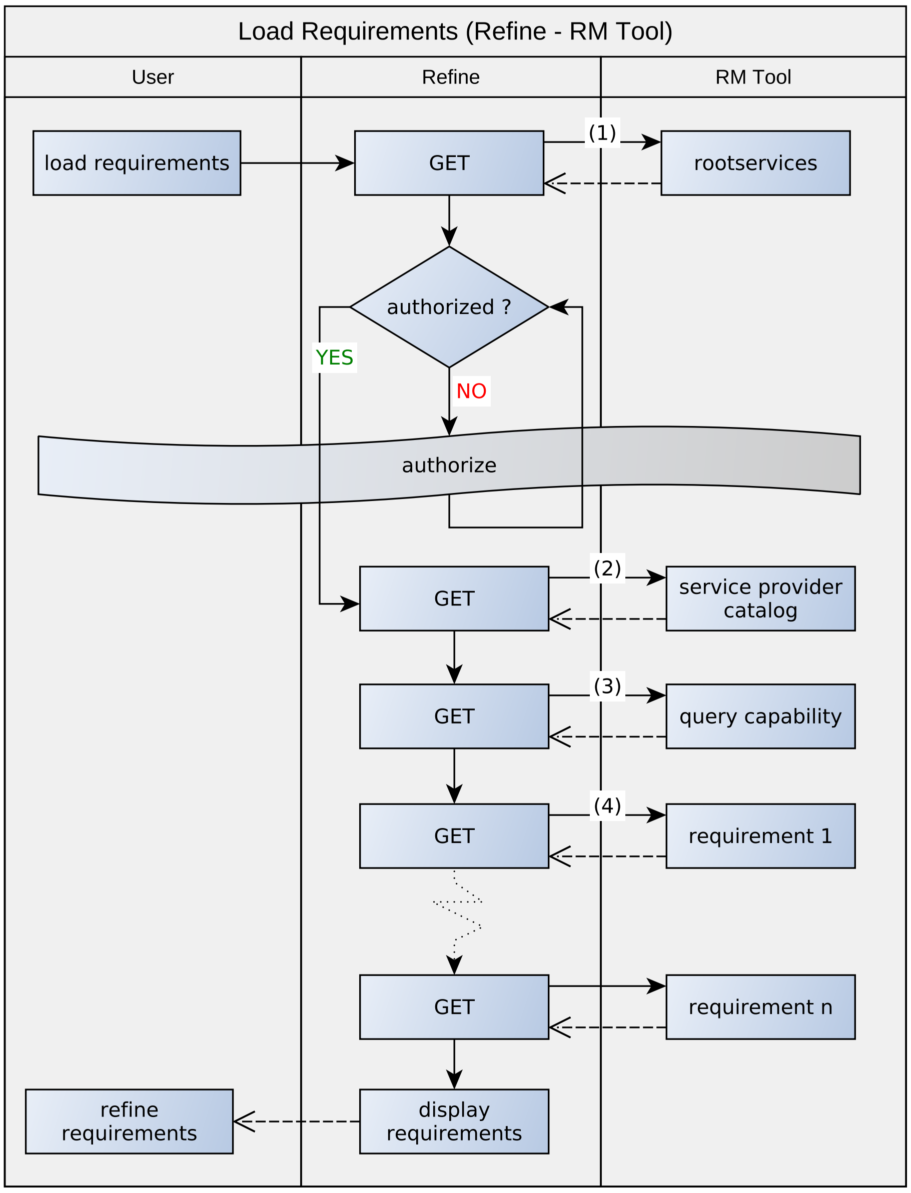


Figure 6.5.: Data flow diagram of Refine and RM Tool

Step 2: The base resource (Listing A.6) is loaded, which may be split up into several pages of base files. The base represents all resources of the TRS provider at a specific point in time. The *cuttoffEvent* corresponds to the point

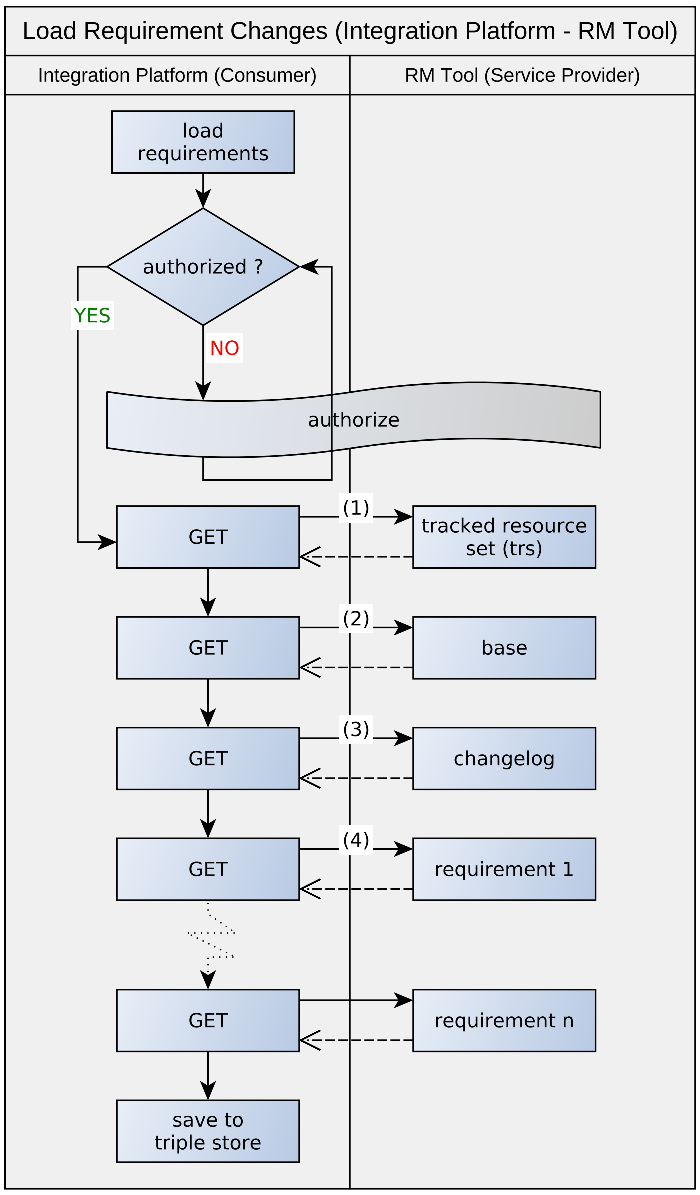

Load Requirement Changes (Integration Platform - RM Tool)
Integration Platform (Consumer)
RM Tool (Service Provider)
load requirements
authorized ?
YES
NO
authorize
GET
(1)
tracked resource set (trs)
GET
(2)
base
GET
(3)
changelog
GET
(4)
requirement 1
GET
requirement n
save to triple store

Figure 6.6.: Data flow diagram of Integration Platform and RM Tool

of time after which modifications to the requirements are not included in the base.

Step 3: Changelogs of modifications, split up to pages, are loaded from RM

Tool. This step is skipped if all changes since the cut-off event are already listed in the TRS document from step 1.

Step 4: Depending on the information from the previous steps, modified or new requirements are loaded from RM Tool using the OSLC READ operation (Listing A.4), and the requirements are stored in an RDF triple store.

6.3. Authorization

The prototype implementation allows authorization with *HTTP Basic*, *OAuth 1.0a*, and *OAuth 2.0* for the OSLC communication between the Integration Platform and RM Tool as well as between Refine and the RM Tool. Those authorization methods were described in Section 4.3. This section describes how the mechanisms were used with the prototype and show the data flow in detail.

6.3.1. HTTP Basic

The simplest form of authorization sends with every single HTTP request an *Authorization* header containing the username and password in an encoded form. The prototype supports HTTP Basic authentication for communication of the Integration Platform and RM Tool, as well as communication from Refine to the RM Tool.

Configuration The Integration Platform and Refine need to know username and password for the RM Tool. At the Integration Platform, an administrator has to configure username and password of a functional user to access the RM Tool. At Refine the user has to configure username and password in the settings of the Refine tool.

Data Flow At each request the Integration Platform or Refine sends to the RM Tool, an authorization header is appended. The header contains username and password for the RM Tool, encoded with Base64.

```
Authorization: Basic base64encoded=un&pw
```

Figure 6.7 gives a schematic view of the data flow.

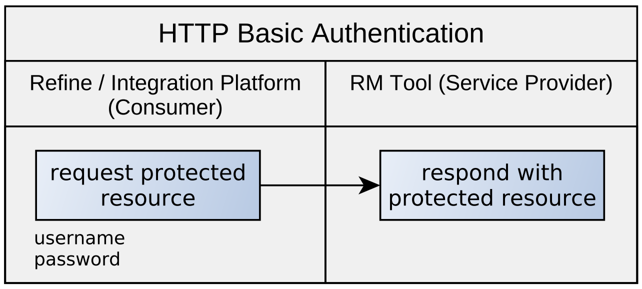

HTTP Basic Authentication
Refine / Integration Platform (Consumer)
RM Tool (Service Provider)
request protected resource
respond with protected resource
username
password

Figure 6.7.: Data flow diagram of HTTP Basic authentication

6.3.2. OAuth 1.0a

Authorization with OAuth allows delegating the authorization to the user while keeping users credentials secret. This means, the client application does not need to know secret user credentials of the service provider. As this form of authorization requires user interaction, it is implemented only for the connection between Refine and the RM Tool.

Configuration No configuration, OAuth URLs are loaded from the OSLC rootservices document.

Data Flow Figure 6.8 shows the data flow for OAuth 1.0a authorization between Refine and the RM Tool in detail. In the first step, Refine uses the *oauthRequestTokenUrl* URL from the rootservices document to get a request token. The HTTP header is signed with an OAuth signature using the `HMAC-SHA1` method with the consumer key.

The second step is the user authorization step. Therefore, Refine calls the *oauthUserAuthorizationUrl* with the received token. Then, the RM Tool redirects the user agent in the browser to an authorization page, requesting an username and password. After a successful authentication, the RM Tool calls the *oauth_callback* URL with the requested token and a unique, randomly-generated verifier.

Finally, in the third step, the authorized request token can be exchanged for an access token. Refine calls the *oauthAccessTokenUrl* with a signed request, containing the token and verifier. The received token secret is appended to the signature. The RM Tool responds with an access token and a token secret.

After completing the three steps to get an access token, the token can be used with a signed request to retrieve protected resources. The service provider determines the validity period of the token.

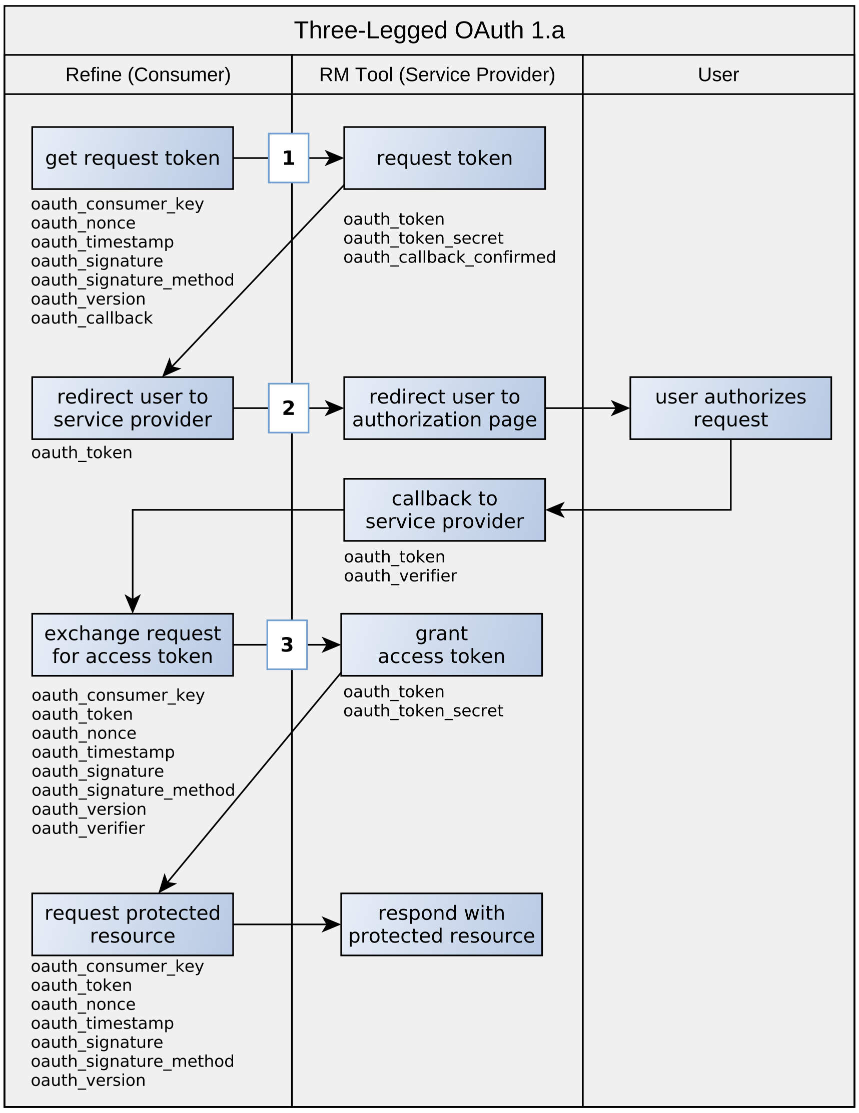


Figure 6.8.: Data flow diagram of OAuth 1.0a

6.3.3. OAuth 2.0 Authorization Code

OAuth 2.0 with grant type *Authorization Code* is implemented to connect Refine with the RM Tool. The configuration and data flow have a huge common ground with OAuth 1.0a.

Configuration The URLs for OAuth 1.0a from the rootservices document can be reused.

Data Flow The data flow drawn in Figure 6.9 is comparable to OAuth 1.0a. The most significant difference is, the messages are not signed.

The process starts by requesting an authorization grant, or authorization code. The *client_id* identifies the client application, *redirect_url* is used for the callback and state can be freely chosen by the client, e.g. to map the request later to the callback. After that, the RM Tool creates a unique authorization code.

In the second step, the client is redirected to a web page for authorization. In the prototype, the RM Tool acts as resource server and authorization server. Therefore the redirection is done instantly after creating the authorization grant. By entering username and password the authorization of the grant is completed and the callback URL of Refine is invoked.

At third, the grant is exchanged for an access token. Therefore the authorization code is sent to the service provider and an access token is issued. The prototype uses token type *bearer*, which is a simple plain text token. Optionally an expiration date and scope for the token could be set.

After completing the three steps to get an access token, the token can be used in an authorization header to retrieve protected resources.

```
Authorization: Bearer access_token
```

6.3.4. OAuth 1.0a Two-Legged

The OAuth flow is called three-legged because there are three steps to receive an access token. Two-legged means to skip one step, the user authorization step. Therefore two-legged OAuth can be used without user interaction, which is required for the connection between the Integration Platform and RM Tool.

The OAuth 1.0a specification [OAu09] does not describe such a two-legged flow. Therefore, different unstandardized variations of two-legged OAuth exist.

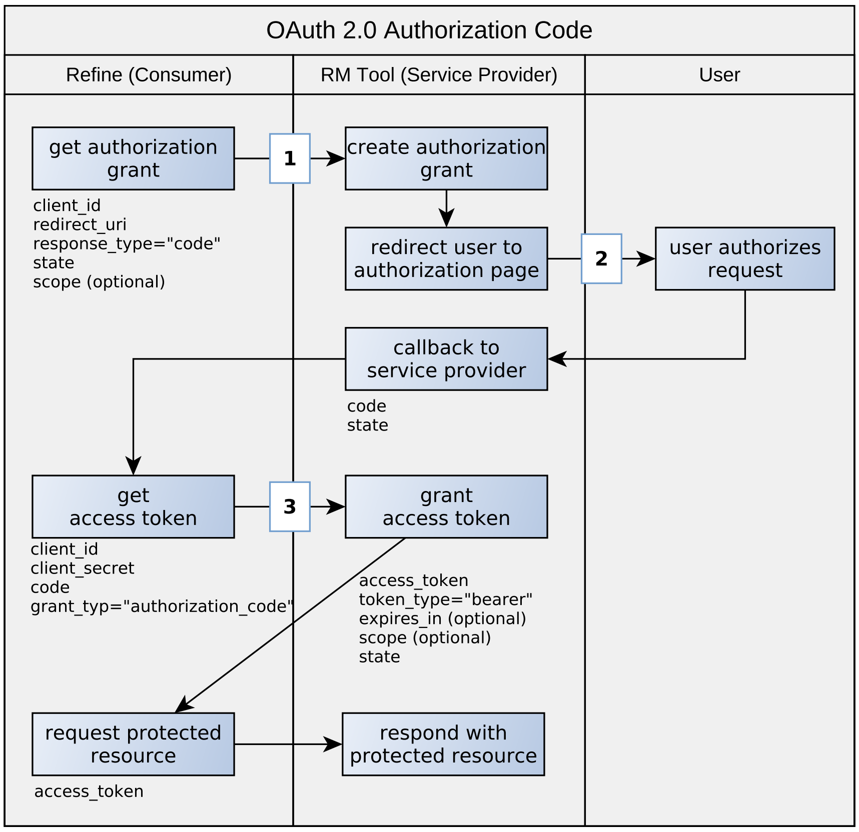


Figure 6.9.: Data flow diagram of OAuth 2.0 with grant type Authorization Code

The username and password of a functional user of the service provider are placed as consumer key and secret of the client. One way is to skip the user authorization step and perform only the steps to get a request token and exchange the request token for an access token.

The prototype uses even a one-step solution, implemented by the Eclipse Lyo libraries. This flow does not require to request access tokens. The request for a resource is signed with the OAuth consumer secret and consumer key. We could call this flow one- or zero-legged, but for convenience, we will also call it two-legged OAuth in the following.

Configuration Consumer key and consumer secret need to be configured at the Integration Platform. Optionally, an OAuth realm has to be set if claimed by the service provider.

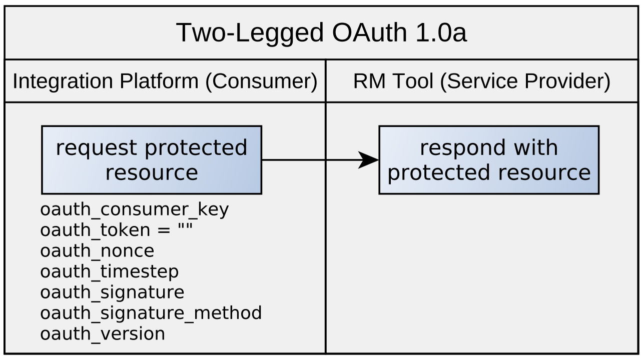


Figure 6.10.: Data flow diagram of two-legged OAuth 1.0a

Data Flow The implemented algorithm skips the OAuth steps to retrieve an access token. Similar to HTTP Basic authentication every OSLC message is signed with the `HMAC-SHA1` algorithm, as described in the OAuth specification [OAu09]. The signature contains an empty *oauth_token*. The data flow is drawn in Figure 6.10.

6.3.5. OAuth 2.0 Resource Owner Password Credentials

In contrast to OAuth 1.0a, the OAuth 2.0 specification [Har12] enumerates authorization grant types which require no user interaction. The *Resource Owner Password Credentials* grant type is used in the prototype in the same way as OAuth 1.0a two-legged. This grant type directly uses the username and password for the *client_id* and *client_secret* and thus can skip the user authorization step. In our scenario, we use the username and password of a functional user of the service provider.

Configuration The client id and client secret need to be configured at the Integration Platform. Furthermore, the URL for requesting an access token is required. This URL may be configured or read from the OSLC rootservices document.

Data Flow As shown in Figure 6.11, there is only one step to get an access token. The token can be used the same way as with grant type Authorization Code.

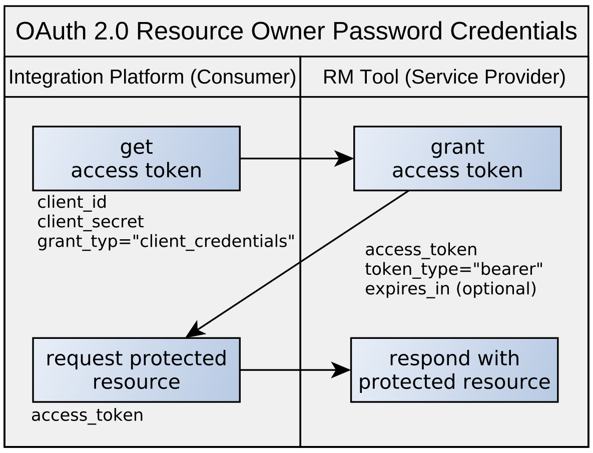

OAuth 2.0 Resource Owner Password Credentials
Integration Platform (Consumer)
RM Tool (Service Provider)
get access token
client_id
client_secret
grant_typ="client_credentials"
grant access token
access_token
token_type="bearer"
expires_in (optional)
request protected resource
access_token
respond with protected resource

Figure 6.11.: Data flow diagram of OAuth 2.0 with grant type Resource Owner Password Credentials

The Integration Platform sends a request with grant type *client_credentials* to the RM Tool. The request contains the *client_id* and *client_secret* which are used as username and password of a functional user of RM Tool. The RM Tool checks the credentials and issues an access token.

6.4. Discussion

The purpose of the prototype implementation was to test different security methods in the defined scenarios. The wide array of different use-cases, methods and technologies forced to focus on a subset of methods.

HTTP Basic authentication was already implemented at all of the components. It's simple usage made it suitable to test communication between the components before implementing the more complex OAuth authorization methods. OAuth 1.0a was implemented throughout the OSLC core specification [osl13] recommends its usage and the Eclipse Lyo library provides functionality for usage on client and server side. Looking ahead in the future, OAuth 1.0a may become obsolete and replaced by OAuth 2.0 and other OAuth related frameworks. Therefore, the prototype was extended with OAuth 2.0 support.

TLS was not deployed with the prototype by purpose. Usage of it makes the configuration of the prototype more complex and hampers analysis of the data flow between the components. Although it is recommended to use in a real environment.

The prototype is evaluated in more detail in the next chapter.

7. Evaluation

In the previous chapter, a prototype implementation was presented. The findings from the prototype are discussed in this chapter.

At the start, in Section 7.1 we discuss which previously defined security objectives are fulfilled by the prototype. Further, Section 7.2 indicates security considerations of the methods and technologies used by the prototype. Finally, the results of the prototype and gained knowledge are discussed in Section 7.3.

7.1. Evaluation Against Objectives

In chapter 3 we defined security requirements based on the RMIAS [CH13] model. The requirements are *authentication*, *authorization and confidentiality*, and *integrity*. The threat analysis in chapter 5 lists and discusses assets and threats and concludes security objectives and countermeasures. The results of this analysis were used to decide about the used methods of the prototype implementation, *HTTP Basic authentication*, *OAuth 1.0a*, and *OAuth 2.0*.

In this section, the prototype implementation is faced against the previous introduced *security objectives*. Those, defined in Section 5.1.5, are: *O.1 Authentication of the server*; *O.2 Authentication of the user*; *O.3 Integrity of OSLC, TRS, and SPARQL communication*; *O.4 Confidentiality of the OSLC, TRS, and SPARQL communication*; *O.5 Grant authorized users permission to create, retrieve, update, and delete OSLC resources*; *O.6 Confidentiality of user credentials*; *O.7 Permission to query via TRS interface*; and *O.8 Permission to query via SPARQL interface.*

The security considerations of the applied methods are omitted in this section but discussed in detail in the next section.

7.1.1. Authentication of the Server

Since HTTP protocols are used as the basis for all parts of the communication, Transport Layer Security (TLS) implemented by HTTPS protocol is the first choice to authenticate the server. By checking the certificate of the server, authenticity of those can be ensured.

A drawback is the administration of the service provider certificates, which have to be installed and updated by the system administrator. This additional administration effort harms the non-functional requirement of simple configuration. Possible solutions are discussed in the security considerations section.

TLS is not activated for the prototype by intention, because the basic architecture stays the same, whatever HTTP or HTTPS is used. On the other hand omitting TLS allowed recording of the communication for development and evaluation purpose.

7.1.2. Authentication of the user

Many lifecycle tools determine access rights to resources by verifying authenticity of the requesting user. The implemented method HTTP Basic authentication can be used to authenticate a user.

However, for the needs of access to resources, instead of an authentication method, a method for delegated access control, such as OAuth, is sufficient. Access control is discussed in the next paragraphs.

For SSO applications authentication is essential, and methods like OpenID Connect may be used therefore. Such a method was not applied for the prototype, but will be outlined in the discussion of this evaluation.

7.1.3. Integrity and Confidentiality of OSLC, TRS, and SPARQL Communication

Integrity and confidentiality are provided when using HTTPS with TLS for all of the communication interfaces. First the consumer or client checks authenticity of the server/provider by verifying the server certificate. Access to sensible or confidential data is given by the provider after an additional step of authorization of the consumer by using the implemented methods HTTP Basic or OAuth. Therefore the whole communication for confidential data is done after mutual authentication. Cryptography of the TLS protocol underlying HTTPS ensures integrity and confidentiality.

7.1.4. Grant Authorized Users Permission to OSLC Operations

The implemented methods HTTP Basic authentication and OAuth are used to control access to resources for authorized users. Confidential or sensible data can only be accessed after a previous authorization of the user. HTTP Basic authentication and OAuth differ in the data flow, as shown in the previous chapter, and therefore provide different levels of security, which will be discussed in the security considerations section. The methods can be used in the same manner by all of the OSLC Create, Retrieve, Update, and Delete (CRUD) operations.

7.1.5. Permission to Query via TRS Interface

The TRS client can authenticate the service provider with TLS. The prototype implementation further facilitates HTTP Basic and OAuth for access control to resources of the service providers. OAuth should be preferred to protect the users credentials against the TRS client application, which was already discussed in the threat and gap analysis.

7.1.6. Permission to Query via SPARQL Interface

HTTP Basic might be used to authenticate the user or OAuth for authentication by the client tool on behalf of the user. However, the Integration Platform needs a way to determine access rights of a user to resources of multiple service providers. As exposed in the conclusion of the threat and gap analysis, the scope of the implementation was limited by omitting SSO and access control methods for the Platform scenario. Possible solutions are discussed in Section 7.3.

7.1.7. Conclusion

Table 7.1 summarizes, which security objectives are covered by the prototype implementation.

Table 7.1.: Security objectives covered by prototype implementation

Security objectives	Prototype implementation
Authe_Server **O.1**	x
Authe_User **O.2**	x1)
Integ_Com **O.3**	x
Conf_Com **O.4**	x
OSLC_Ops **O.5**	x
Conf_Cred **O.6**	x2)
TRS_Query **O.7**	x
Sparql_Query **O.8**	∼

x covered, ∼ partly covered

1) If HTTP Basic authentication is used.

2) If OAuth is used.

The prototype implementation covers all security objectives related to the OSLC scenario. Confidentiality of the user's credentials can be assured by using OAuth 1.0a or OAuth 2.0 instead of HTTP Basic authentication. When we further assume exclusive use of HTTPS, the solution can further provide authenticity of the service provider, integrity and confidentiality of the communication channel, as well as access control for OSLC and TRS interfaces. In conclusion the prototype is suitable to cover these objectives. In the next section, security considerations of the applied methods are discussed. Open issues to cover the objective of access control for SPARQL interfaces of the Integration scenario are treated in the discussion of this chapter and the previous threat and gap analysis.

7.2. Security Considerations of Applied Methods

All of the security methods from the implementation infer different security issues. In Chapter 5 we discussed various threats, including *disclosure of communication*; *message insertion/modification/deletion*; *impersonation of identities*; *disclosure of user credentials*; *unauthorized access*; and *API misuse*. Now we debate concrete security issues of the prototype and elaborate security considerations which have to be made for an implementation in a real environment.

TLS is mandatory for a proper communication channel protection, therefore we start with TLS before talking about the implemented authentication/authorization methods. A basic description of the methods can be found in Chapter 4.

7.2.1. Transport Layer Security (TLS)

In the LTI scenario, TLS is used as an underlying transport layer of HTTP. [Res00] illustrates the resulting HTTP protocol. The first source for security considerations to TLS is the official specification of the protocol [DR08]. [MS13] provides a historical overview of attacks on different versions of TLS.

Protocol Version. To allow legacy support, browsers and other systems support connections to former SSL/TLS protocols. The SSL protocol, in version 1.0, 2.0, and 3.0 was obsoleted by TLS because weaknesses have been found; hence it should not be used any longer. Most browsers stopped support for SSL 3.0 after the *Padding Oracle On Downgraded Legacy Encryption (POODLE)* attack was exploited [MDK14]. POODLE makes man-in-the-middle attacks to obtain data via a padding-oracle-attack. Doung and Rizzo introduced *Browser Exploit Against SSL/TLS (BEAST)* [DR11], an chosen-plaintext attack against SSL 3.0 and TLS 1.0. Therefore e.g. the U.S. National Institute of Standards and Technology (NIST) recommends using TLS 1.1 or 1.2 over TLS 1.0 [PMC14].

Currently, TLS 1.1 and TLS 1.2 provide the highest level of security. Important is to keep in mind, limiting the set of supported versions reduces interoperability with (older) software systems. Contrariwise, older version open gateways to offenders for eavesdropping and MITM attacks.

Cipher Suite TLS supports various cipher suites. The cipher suite defines a method for key exchange, a bulk encryption method, a message authentication code, and a pseudorandom function. An example is `TLS_RSA_WITH_AES_128_CBC_SHA`, with `RSA` for key exchange, `AES_128_CBC` as 128-bit symmetric block encryption method in `CBC` mode, and `SHA-1` as the hash function. Server and client negotiate which cipher suite is used. But for both, a system administrator can limit the set of allowed cipher suites. Selection of used cipher suites directly influences the security level, e.g. `AES_128` has a complexity of 2^{128}, while recovering the block cipher `3-DES` has been proven with a complexity of 2^{86} [Bih96]. Recommendations to cipher suites can be looked up from different institutes, an overview of those is given by [SKV^{+}16].

Usage. The entire content provided by the service provider must be protected with TLS. Intuitively, URLs to protected resources need to be secured via TLS, otherwise eavesdropping in a physical network (LAN) [JZI10] or a wireless network (WLAN) [Cas02] is an easy task. But further other URLs, like rootservices, OSLC service provider catalogs, or OAuth URLs, have manifold effects on security. If an attacker can modify URLs inside a rootservices or service provider catalog document, the attacker can redirect consumers to malicious pages. Accordingly, unprotected pages offer possibilities for attackers to read and modify content by MITM attacks or redirect clients unnoticed to fake service providers.

Certificate and Key Management. Certificates, including the key material used to verify authenticity and establishment of a confidential connection, are a critical part of the architecture.

> "The system is only as strong as the weakest key exchange and authentication algorithm supported." [DR08]

Recommendations from NIST, according to TLS algorithms and key lengths can be found in [PMC14]. Key generation tools typically generate certificates with of sufficient strength. For instance, the OpenSSL tool generates keys for RSA with a default key length of 2048 bits.

> "Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage." [DR08]

A critical part in LTI is limiting acceptable certificates to the smallest possible amount, considering usability and system administration efforts. In the best case, a client tool allows only connections to a list of registered service providers, and vice versa. The requirement for lightweight integration makes such an approach impractical, due to the administration effort of maintaining records of trusted service providers per lifecycle tool. The TLS specification of X.509 certificates [CSF$^+$08] describes the concept of certificate chains. These chains can be used to arrange a lifecycle with different levels of trust, as described in Section 4.1. By example, a corporation can create a root certificate which is trusted by all clients within the organization. For more granularity, a corporation can issue

Certificate Authorities (CAs) on project levels, whereas lifecycle tools trust only certificates issued with these CA certificates. On department, corporation, and other trust levels, similar solutions can be deployed.

User Awareness

> "Even when certificates are implemented "perfectly" human behavior often renders them moot." [Dav11]

All browsers at the time allow users to ignore certificate warnings for mismatched domains or certificates past its validity time [Dav11]. With such behavior, a user might connect to a fake service provider or the communication is intercepted by a malicious proxy server. Where feasible, client tools should remove the possibility to accept invalid certificates by the users, for browser-based clients education of user awareness is vital.

Faulty Configuration or Implementation. Accepting invalid or unknown server certificates entirely breaks the TLS protection. But a proper implementation and configuration of TLS can be time-consuming. Especially in non-browser software, the tool developers need to handle certificate validation. [GIJ$^+$12] demonstrate broken SSL/TLS implementations for non-browser software. In the worst case, a developer disables certificate validation to counter problems with certificates. [GIJ$^+$12] lists some quotes from developer's forums:

> "I want my client to accept any certificate (because I'm only ever pointing to one server) but I keep getting a `javax.net.ssl.SSLException: Not trusted server certificate exception`"[1]

7.2.2. HTTP Basic Authentication

Although HTTP Basic Authentication is easy to implement with a high interoperability, Basic Authentication is *"[...] very much on the weak end of the security strength spectrum"*. [FHBH$^+$99a]

[1]http://stackoverflow.com/questions/2642777/trusting-all-certificates-using-httpclient-over-https (Accessed: 2016-11-25)

Digest Authentication. Using weak passwords, Basic Auth is vulnerable to dictionary and brute force attacks. An improvement to Basic Auth is Digest Auth, also described in the same protocol specification [FHBH$^+$99a]. In general Digest Auth uses hashed instead of plaintext passwords for transfer. If sufficiently strong cryptographic hash algorithm are used, plaintext passwords are not recoverable from the hash values. Compared to other authentication mechanisms, Digest Auth is still considered to have a weak security level. The specification [FHBH$^+$99a] lists scenarios where MITM attacks are possible for Basic and Digest Auth, [ANN03] further describes MITM attacks even when tunnelling the communication via TLS.

However, Digest Auth solves many problems of Basic Auth, such as brute force, dictionary attacks, eavesdropping of passwords, and replay attacks. But the security level depends on the usage of a hash function. The specification describes only the use of MD5 [Riv92]. This message-digest algorithm has already shown several security issues [WFLY04], and even the author of MD5, Ronald L. Rivest, stated it as *"clearly broken (in terms of collision-resistance)"* [Riv05]. A hash function is *collision resistant* if it is difficult to find the inputs to the hash function which result in the same output. Applying such an attack to Digest Auth is not practical at the time, but might be feasible conceivable.

Storing Passwords. When using Basic Auth for web applications, the users browser stores the credentials in place of the client application. In LTI different types of applications, such as native or browser-based, might require storing the credentials by the client tool. In the prototype both, Refine and the Integration Platform, store username and password to gain access to the RM Tool. In a more complex scenario with multiple OLSC/TRS consumers and providers, each of them needs to store client credentials of each other, leading to a massive security issue.

Every client tool accessing a service provider has to store user credentials from the provider. E.g., if ten applications access a service provider, ten applications hold the passwords, additionally to the service provider. The likelihood of compromised passwords increases dramatically.

Client Tool Authentication. In the LTI scenarios, we need to authenticate the service provider and the client tool. As described in the last paragraphs, TLS with server certificates can be used to verify the authenticity of the service provider. If deploying certificates for the client tools is feasible, those can be used to verify the client tools' authenticity.

Otherwise, the user authentication/authorization method can be considered as a solution. When utilizing OAuth, the service provider identifies the client tool by a unique pair of client identifier and client secret. Using a compromised OAuth access token is not possible as long as the attacker cannot present a valid client identifier and secret to the service provider. There exists no similar concept for Basic/Digest Auth, accordingly an attacker can immediately use compromised user credentials for requests to a service provider.

7.2.3. OAuth 1.0a

Similar to HTTP Basic Authentication, OAuth 1.0a is foreseen by the OSLC Core Specification Version 2.0 [osl13]. Additionally, we assume to use TLS as underlying transport layer protection protocol. The first source for security issues of the OAuth 1.0a protocol is the security considerations chapter of the specification [OAu09].

Protocol Version. For OAuth 1.0 a security flaw called session fixation attack [Ham09] was published, which led to the development of OAuth 1.0a. The attacker connects to the service provider to receive a request token. Instead of authorizing it, the attacker convinces the victim by social engineering to click a link for authorizing the token. The service provider accepts the request, as the request token is valid. The victim cannot detect an attack is ongoing, as the victim is redirected to the legitimate page of the service provider. The attacker now exchanges the request token to an access token. Finally, the attacker can use the approved access token to request resources with access rights of the victim.

For mitigation of those attack, an OAuth 1.0a client sends the *oauth_callback* URI with the query of the request token. An attacker cannot modify these URI. After user authorization, the service provider invokes the callback and appends an *oauth_verifier*. This verifier is necessary to exchange the request token for an access token, but the attacker is not able to read the response from the callback.

Since TLS does not prevent the session fixation attack, OAuth 1.0a must be used. The following security considerations hold for OAuth 1.0a.

Signature. OAuth specifies three different signature methods, `HMAC-SHA1`, `RSA-SHA1`, and `PLAINTEXT`, which are used to sign all OAuth requests. A signature is not necessary if the protection with TLS holds, but it provides additional security if TLS is disabled or broken. `PLAINTEXT` appends a signature string without the use of a cryptographic operation, it does not increase the level of security. Utilizing `PLAINTEXT` signatures in combination with TLS could be considered, but `HMAC-SHA1` or `RSA-SHA1` should be preferred. The prototype uses `HMAC-SHA1` signatures for OAuth 1.0a requests.

A further advantage in security is that the signature binds an access token to a relying party. Only the consumer tool which claimed the OAuth token can use it since every request to protected resources needs to be signed with the consumer secret and token secret.

```
Signature key: oauth_consumer_secret&oauth_token_secret
```

The token secret is unique for each OAuth token; the consumer secret is unique for each client tool. Hence the access token is bounded for usage only by the requester of the access token. A compromised access token cannot be used by an attacker unless the consumer secret is compromised as well. Breaking TLS is not sufficient to break OAuth 1.0a, the consumer secret needs to be compromised by some additional attacks.

The consumer secret has to be protected by the service provider and consumer. The service provider is typically a web server application and therefore able to store the credentials in a protected database on the server. Protection by the consumer is more challenging and depends on the type of the client. Possible solutions are to store the credentials in a hardware security module (HSM) or to query the credentials from a web service. Obfuscating the credentials in the source code should be only considered for tools with low-security demands because decompiling the application to retrieve the secret is just a matter of time.

Scoping. The protocol defines no ways to scope the delegation rights. The service provider may want to restrict delegated access rights of a consumer tool to specific domains. E.g., a TRS client could be limited to have read-only access rights.

Clickjacking and Automatic Login. Malicious consumer tools can request OAuth tokens without notice of the user by a clickjacking [HMW^{+}12] attack.

The attack is done in the following steps:

1. The malicious tool M gets a new request token from the service provider SP.
2. M convinces a user U of SP to click a link by a social engineering attack.
3. The user agent of U is redirected to a malicious page of M. The page contains the user authorization page of SP as hidden iFrame covered by another frame of M. Figure 7.1 shows a demonstration page. A fake page displaying cat photos covers the OAuth authorization page to the *RMProvider*. If U clicks to see more cats, U authorizes a token for M to access the RM Tool.
4. M convinces U to click on a link of the page. If the user clicks the link, instead the *accept* button of the hidden iFrame is clicked.
5. The token of M was authorized without knowledge of U. M can access protected resources on behalf of U.

There are several ways to prevent a web page is displayed within an iFrame. Framebuster, also called framekiller, are JavaScript implementations which detect if the page is the top level window or embedded in another page. Framebasters can disable navigation if the authorization page is embedded. Though there are solutions to bypass framebusting [RBBJ10]. Some browsers allow to take advantage of the `X-FRAME-OPTIONS` header to block any iFrames.

An effective and general solution is, to deactivate automatic login. The user is asked by the service provider for the user credentials with every token authorization request. Even if the user has an open session at the service provider or already requested a token for the client earlier, submission of user credentials is mandatory. Figure 6.2 shows a web page of the prototype implementation, asking the user for authorization of an access request. The Eclipse Lyo [ecl16] library for OAuth implements a session management and only asks for a password if the user has no session open. For improved security, that behaviour should be changed.

Two-Legged. The OAuth 1.0a protocol specification defines no way for delegated access without user interaction. Our use-cases require such a mode for the connection of TRS interfaces. The solution for the prototype was a derivative from the specified three-legged flow. The algorithm is described in Chapter 6, and the data flow is pictured in Figure 6.10.

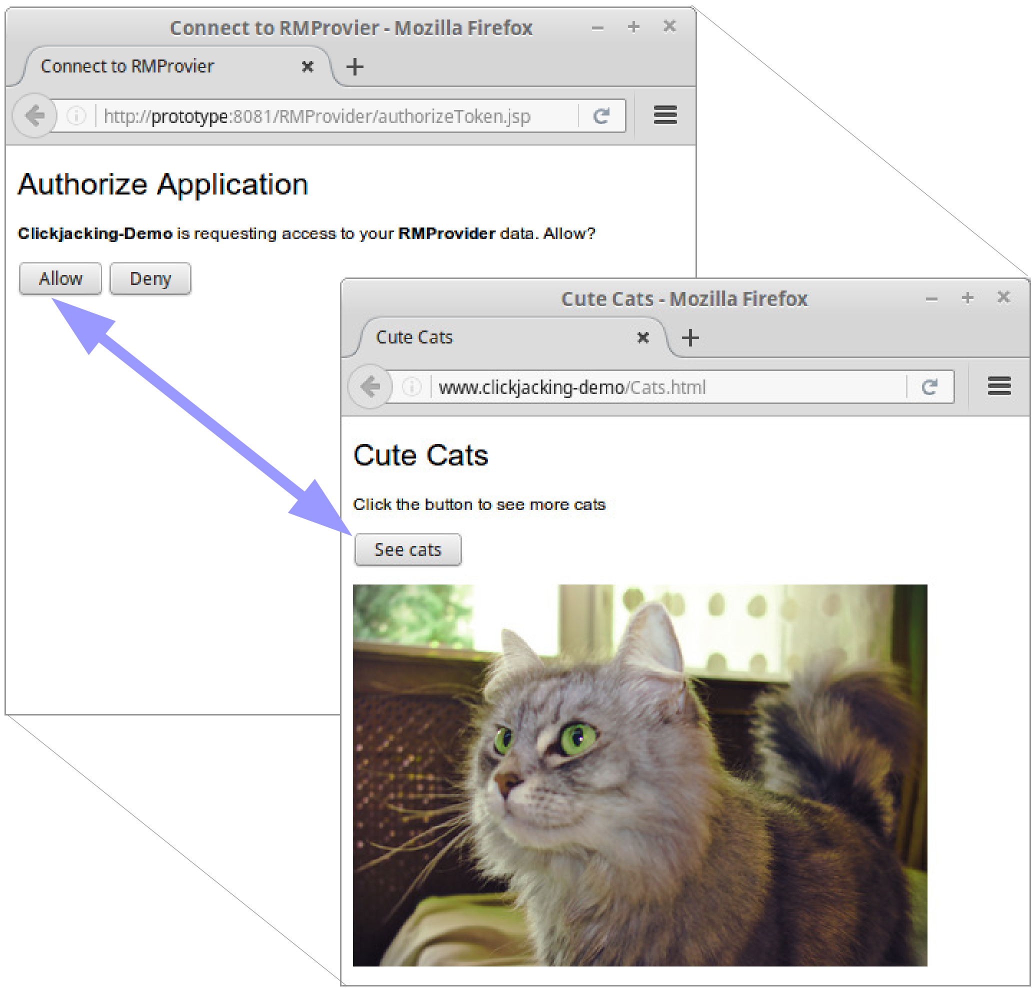

Connect to RMProvier - Mozilla Firefox
Connect to RMProvier
http://prototype:8081/RMProvider/authorizeToken.jsp
Authorize Application
Clickjacking-Demo is requesting access to your RMProvider data. Allow?
Allow
Deny
Cute Cats - Mozilla Firefox
Cute Cats
www.clickjacking-demo/Cats.html
Cute Cats
Click the button to see more cats
See cats

Figure 7.1.: Demonstration of clickjacking of OAuth 1.0a

The two-legged mode is not specified, but the implementation is straight forward if the three-legged mode already is used. However, due to the lack of a specification, service providers and clients may implement the method slightly different resulting in interoperability problems and questionable security.

7.2.4. OAuth 2.0

The intention on the development of OAuth 2.0 [Har12] was to simplify the OAuth 1.0a signature process and adapt it for multiple devices. OAuth 1.0a showed more and more problems with native and mobile applications. The IETF has published a threat model and security considerations [LMH13] for OAuth 2.0.

OAuth 2.0 and the Road to Hell. In 2009, Eran Hammer, author of OAuth 1.0 and at this time lead author of OAuth 2.0, resigned and withdrew his name from the OAuth 2.0 framework specification. He explained his reasons in a blog writing called "OAuth 2.0 and the Road to Hell" [Ham12] claiming the specification is likely to produce insecure implementations.

OAuth 2.0 defines multiple grants for usage with different client types, removes signatures and introduces the concept of *bearer token*. Chapter 6 describes bearer tokens and some of the grant types. Removing signatures should help to make the client side implementation easier. Further, the grant types specify flows for different application types, including native and mobile applications. At least the framework is easily extensible. Multiple protocols, including OpenID Connect and UMA, are built on top of OAuth 2.0.

The main points of Hammer are: security solely depends on the correct implementation and usage of TLS; tokens are not bound to a consumer tool; and the specification is much more comprehensive, harder to understand for implementers without a solid security background knowledge.

Unbound Tokens. A critical issue of OAuth 2.0 are the unbound access tokens, namely the bearer tokens. A token can be leaked by several ways, on transport, at consumer or service provider, and at authorization server. In difference to OAuth 1.0a, a leaked OAuth 2.0 bearer (access) token is sufficient to access resources, without the need to steal the consumer secret. The bearer token specification [JH12] provides some recommendations for their usage, as the usage of TLS, and short-term validity durations of the tokens. Further, it recommends to add authentication information of the client application to the token but does not specify how.

A possible solution is the generation of signatures, similar to OAuth 1.0. Additionally, binding of bearer tokens to clients could prevent utilization of leaked token.

Implementation Challenges. Implementing an OAuth 2.0 service provider is challenging when considering all possible vulnerabilities. On client side things are much easier, nevertheless, using tested libraries is recommended. The prototype uses the Apache Oltu library [Fou16a], but especially when implementing a service provider, still many vulnerabilities need to be considered.

Yang and Manoharan identified six root causes of vulnerabilities at their study of security vulnerabilities of the OAuth 2.0 protocol [YM13].

1. *No requirement or no recommendation of TLS protection on callback endpoints.*
   TLS needs be used for all endpoints, including the callback endpoint of the client. This requirement complicates the development of the client tool, but provides authentication of the client callback endpoint and reduces risks of the leakage of an authorization code.
2. *Allowing multiple uses of authorization codes.*
   To overcome this problem, limiting the validity of authorization codes is requested by the specification. It requires to limit validity time of the codes to ten minutes and recommends to allow usage of a code only once. At an attempt to use an authorization code multiple times, the authorization server should revoke all access tokens issued previously with this authorization code.
3. *The removal of the signature requirement of authorization request disables authorization server to validate the authenticity of the client application.*
   Optionally, a client tool can be authenticated with TLS, but this solution might be impractical due to the additional effort of certificate management, as discussed before.
4. *No vetting process is enforced to ensure the security of the client application before enabling the automatic authorization granting feature.*
   As discussed for OAuth 1.0a *Clickjacking and Automatic Login*, entering user credentials for every token request should be enforced. But forcing the user to type in credentials declines the usability. Instead the service provider may verify additional information, e.g. the client's IP address, and ask the user only for a password if these information cannot be verified.
5. *Flexible redirection URIs validation mechanism is not adopted.*
   OAuth 2.0 uses client callback URIs to bind the client to the requested token. Flexible URIs allow to manipulate them while the validation is still valid. For example, an attack may manipulate a relative URI to redirect callback from authorization server to a malicious location. Such an attack

is called open redirector or covert redirect [BBDLM14].

6. *No authenticity of the authorization code.*
   Authorization codes are mapped to a redirect URI, but not to an authenticated client. OAuth 1.0a establishes this authentication with signatures. The problem is similar to unbound tokens as discussed before.

Scopes OAuth 2.0 allows defining scopes for access tokens. By example, tokens for TRS clients may be limited to read-only rights.

Two-Legged. On the contrary to OAuth 1.0, the OAuth 2.0 framework specifies flows, called grants, for delegated access without user interaction. The prototype uses the *Resource Owner Password Credentials* grant, described in Section 6.3.5.

A *client_id* and a *client_secret* are used for authenticating the client tool and making the access decision by the authorization server. Because the secret is transferred in plaintext, usage of TLS is mandatory. Each tool with two-legged access, e.g. TRS clients, have to be registered at the service provider with the *client_id*. The service provider may link a functional user[2] to the *client_id* for handling access rights similar to regular users. Each client tool should be registered with an unique *client_id*. Security of the methods depends on the service provider and client tools to keep the *client_secret* confidential.

Comparison of OAuth 1.0 and OAuth 2.0 Table 7.2 gives a brief comparison of OAuth 1.0 and OAuth 2.0 from Prabath Siriwardena [Sir14]. The differences from the evaluation of the prototype implementation are described in the following.

Siriwardena used the hash algorithm `SHA-256` instead of the algorithm specified [OAu09] for OAuth 1.0a, `SHA-1`. The `SHA-256` algorithm provides higher security than `SHA-1`, but using unspecified algorithms causes problems in interoperability. Further, the comparison declines OAuth 1.0 as less developer friendly. That is a typical statement, based on the effort for creating signatures. But implementation of the prototype pointed out that in fact, the signatures

[2]A functional user is utilized by a client tool to access resources of a service provider, with access rights of the functional user. The service provider maps the functional user uniquely to the appropriate client tool. Thereby, a client tool can access resources of a service provider without the need of authenticating with a normal user.

Table 7.2.: OAuth 1.0 vs. OAuth 2.0 from [Sir14]

OAuth 1.0	OAuth 2.0
An access-delegation protocol	An authorization framework for access delegation
Signature based: `HMAC-SHA256`, `RSA-SHA256`	Non-signature-based, Bearer Token Profile
Less extensibility	Highly extensible via grant types and token types
Less developer friendly	More developer friendly
TLS required only during the initial handshake	Bearer Token Profile mandates using TLS during the entire flow
Secret key never passed on the wire	Secret key goes on the wire (Bearer Token Profile)

require no effort, as libraries can be used to create them. On the other hand, OAuth 2.0 is a much more comprehensive protocol, especially for implementation of the authorization and resource server the effort is much higher. Furthermore, Siriwardena states that TLS is required only during the initial handshake of OAuth 1.0. It is true that signatures protect integrity and authenticity of access tokens, but TLS is still required to protect integrity and confidentiality of the payload and response from requests.

7.3. Discussion and Recommendations

Concluding from the evaluation we discuss and recommend the methods for the scenarios introduced in Chapter 2.

Security of the Communication Channel. Certain is TLS needs to be deployed for service providers. This holds for all types of interfaces, as OSLC, TRS, or SPARQL service providers interfaces. The evaluation indicated especially the need to protect OAuth callback endpoints via the HTTPS protocol [YM13]. An issue is the effort for certificate and key management; the evaluation discusses an approach to organize server certificates in levels of trust, based on certificate chains.

Authorization Both integration scenarios rely on access control between the tools. HTTP Basic authentication was evaluated because it is recommended by the OSLC specification. Security of this authentication form was shown to be weak. Digest authentication, which transfers message digests of the user credentials instead of plaintext credentials, is an improvement but still threatens confidentiality of user credentials.

In contrast, OAuth 1.0a keeps the user credentials confidential towards the consumer tools. Furthermore, the specification is easy to understand, and the signatures add an additional security level compared to HTTP Basic authentication or OAuth 2.0. But just those signatures complicate development of OAuth 1.0a clients, a disadvantage which can be tempered using libraries as shown with the prototype implementation. Other disadvantages are the lack of access scopes, flows for different client types, and the missing flow without user interaction. But the key argument against OAuth 1.0a is the future development. The most prominent service providers, such as Amazon, Google, Facebook, or Microsoft, have already switched to OAuth 2.0. Using OAuth 1.0a for legacy support may be convenient, but new tools may already have implemented OAuth 2.0 more likely.

In conclusion from the prototype and evaluation, we recommend to favor OAuth 2.0 over OAuth 1.0a and HTTP Basic authentication. However, the OAuth 2.0 framework has some security issues, if the implementation of client or server is not in proper form and does not consider all vulnerabilities. Libraries could be used to decrease the risk, but as far as we know, there is no reference implementation of an OAuth 2.0 server and client for an OSLC service provider and consumer. We recommend to extend the Eclipse Lyo libraries [ecl16] for OSLC and TRS interfaces by an OAuth 2.0 implementation to ease the development of new tools in a lifecycle. The prototype implementation demonstrates such an implementation. Due to the similarities to OAuth 1.0a, OAuth 2.0 can be implemented for OSLC without the need of modifications to the OSLC core specification.

Future work For the Platform scenario (Section 2.3.2) SSO is a useful requirement. OpenID Connect is based on OAuth 2.0, and therefore an convenient adoption. The OpenID Connect protocol reuses the methods from OAuth 2.0 for delegated access, and includes the functionality of authentication. All connected tools need to use the same identity server or synchronize the user information. In an corporate environment a LDAP service might be used. Otherwise protocols as SCIM might be considered. The prototype implementation does not cover SSO and federation of identities, therefore further research is required for the Platform scenario.

Another point of the Platform scenario which requires more investigation is access control to the SPARQL service of a TRS client. Possible solutions were discussed in the threat and gap analysis of Section 5.2.

8. Conclusion

The process of securing lifecycle tool integrations is subject to diverse security requirements spanned by various integration scenarios. Therefore, one of the first and most important steps of this work was to define relevant integration scenarios and infer security requirements.

The OSLC [osl13] scenario covers the communication between tools in the lifecycle. The Platform scenario extends the communication scenario by a central integration platform used to construct a common database of OSLC resources. Querying the database is possible without the need of contacting the resource holding tool. Our derived security requirements include authentication of tools and users, authorization to access resources, confidentiality of resources and user credentials, and integrity of data in transit.

A threat and gap analysis was used to identify suitable methods utilized for a prototype implementation. The architecture of the prototype involves three tools including an OSLC consumer, an OSLC service provider, and a TRS [osl15b] client regularly loading resources from service providers. HTTP Basic authentication [FHBH+99a], OAuth 1.0a [OAu09], and OAuth 2.0 [Ham12] were used to comply the security requirements. The final evaluation includes a comparison of the utilized methods and gives security considerations as well as recommendations of methods to deploy for the scenarios.

Summarized, OAuth 2.0 satisfied the needs for the OSLC communication parts most. In combination with TLS for authentication of the tools, as well as protection of integrity and confidentiality of the communication channels, OAuth 2.0 fulfills all requirements for delegated access to resources. In comparison to OAuth 1.0a, it can be considered as less secure because of the missing signatures, however, it supports multiple devices and data flows, is adjustable and can be used as an underlying protocol for multiple other specifications. This makes it to the most promising access delegation protocol of the next years. In the evaluation we recommend to extend the libraries [ecl16] for OSLC by OAuth 2.0, similar to the work we have done in the prototype implementation.

Further research is necessary for the part of access control to the integration platform. Different solutions were discussed in the threat and gap analysis, as well as in the discussion of the evaluation. A promising candidate for a lightweight integration is a centralized access control model encapsulated from the tools, but with disadvantages due to the redundant access control policies. A more comprehensive solution, suitable for organizations with large lifecycles is the usage of the XACML [OAS13] protocol. UMA [HMMC15] might be considered to combine delegated access of OAuth 2.0 with access control using XACML.

To close with a remark from the introduction, security should be thought as an integral part when linking tools in a lifecycle.

> *Security should never be an afterthought - it's an integral part of any software system design, and it should be well thought out from the design's inception.* [Sir14]

Appendix A. Sample Messages

This appendix lists sample OSLC and TRS messages in XML format. Irrelevant parts of the messages for gaining an understanding of OSLC and TRS were removed to improve readability.

A.1. OSLC Messages

OSLC Rootservices

HTTP GET http://prototype:8081/RMProvider/rootservices

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:Description rdf:about="http://prototype:8081/RMProvider/rootservices"
  xmlns:oslc_rm="http://open-services.net/ns/rm#">
        <dcterms:title>OSLC-RM Adapter/RM Provider Root Services</
            dcterms:title>
        <oslc_rm:rmServiceProviders rdf:resource="http://INNB01637:8081/
            RMProvider/services/catalog/singleton" />
        <jfs:oauthRealmName>RMProvider</jfs:oauthRealmName>
        <jfs:oauthDomain>http://prototype:8081/RMProvider/</jfs:
            oauthDomain>
        <jfs:oauthRequestTokenUrl rdf:resource="http://prototype:8081/
            RMProvider/services/oauth/requestToken"/>
        <jfs:oauthUserAuthorizationUrl rdf:resource="http://prototype
            :8081/RMProvider/services/oauth/authorize" />
        <jfs:oauthAccessTokenUrl rdf:resource="http://prototype:8081/
            RMProvider/services/oauth/accessToken"/>
```

```
</rdf:Description>
```

Listing A.1: RM Tool Rootservices

OSLC Service Provider Catalog

HTTP GET http://prototype:8081/RMProvider/services/catalog/singleton

```
<rdf:RDF
  xmlns:oslc="http://open-services.net/ns/core#">
    <oslc:ServiceProviderCatalog rdf:about="http://prototype:8081/
        RMProvider/services/catalog/singleton">
        <oslc:domain rdf:resource="http://open-services.net/ns/rm#"/>
        <oslc:serviceProvider>
            <oslc:ServiceProvider rdf:about="http://prototype:8081/
    RMProvider/services/serviceProviders/1">
                <oslc:prefixDefinition>
                    <oslc:PrefixDefinition>
                        <oslc:prefixBase rdf:resource="http://open-services.
    net/ns/core#"/>
                        <oslc:prefix>oslc</oslc:prefix>
                    </oslc:PrefixDefinition>
                </oslc:prefixDefinition>
                <dcterms:title rdf:parseType="Literal">Beates RM Service
                     Provider</dcterms:title>
                <oslc:details rdf:resource="http://prototype:8081/RMProvider/
                    services/"/>
                <oslc:service>
                    <oslc:Service>
                      <oslc:queryCapability> <!--Read Requirements-->
                            <oslc:QueryCapability>
                                <oslc:resourceShape
                                        rdf:resource="http://prototype:8081/
    RMProvider/services/requirement?resourceShapes=true"/>
                                <oslc:queryBase rdf:resource="http://
                                    prototype:8081/RMProvider/services/
                                    requirements"/>
                                <oslc:usage rdf:resource="http://open-services
                                    .net/ns/core#default"/>
                                <oslc:resourceType rdf:resource="http://open
                                    -services.net/ns/rm#Requirement"/>
                                <oslc:resourceType rdf:resource="http://open
                                    -services.net/ns/rm#LinkType"/>
                                <dcterms:title rdf:parseType="Literal">
                                    Resource Provider Query Capability
                                </dcterms:title>
                                <oslc:label>Resource Provider Query
                                    Capability</oslc:label>
                            </oslc:QueryCapability>
                        </oslc:queryCapability>
                        <oslc:creationFactory> <!--Create Requirements
                            -->
                            <oslc:CreationFactory>
                                <oslc:resourceShape
                                        rdf:resource="http://prototype:8081/
    RMProvider/services/requirement?resourceShapes=true"/>
                                <oslc:creation rdf:resource="http://prototype
                                    :8081/RMProvider/services/requirements"/>
                                <oslc:usage rdf:resource="http://open-services
                                    .net/ns/core#default"/>
                                <oslc:resourceType rdf:resource="http://open
                                    -services.net/ns/rm#Requirement"/>
                                <dcterms:title rdf:parseType="Literal">
                                    Resource Provider Creation Factory
                                </dcterms:title>
                                <oslc:label>Resource Provider Creation
                                    Factory</oslc:label>
                            </oslc:CreationFactory>
                        </oslc:creationFactory>
                        <oslc:domain rdf:resource="http://open-services.net/
                            ns/rm#"/>
                    </oslc:Service>
                </oslc:service>
                <dcterms:identifier>1</dcterms:identifier>
                <dcterms:description rdf:parseType="Literal">OSLC
    Service Provider for RM service</dcterms:description>
```

```
                    <dcterms:created rdf:datatype="http://www.w3.org/2001/
                        XMLSchema#dateTime">2016−07−22T12:11:53.636Z
                    </dcterms:created>
                </oslc:ServiceProvider>
            </oslc:serviceProvider>
            <dcterms:description rdf:parseType="Literal">OSLC Service
                Provider Catalog</dcterms:description>
            <dcterms:title rdf:parseType="Literal">OSLC Service Provider
                Catalog</dcterms:title>
        </oslc:ServiceProviderCatalog>
    </rdf:RDF>
```

Listing A.2: RM Tool Service Provider Catalog

OSLC Query Capability

HTTP GET http://prototype:8081/RMProvider/services/requirements

```
<?xml version="1.0" encoding="UTF−8"?>
<rdf:RDF
    xmlns:oslc="http://open−services.net/ns/core#"
    xmlns:oslc_rm="http://open−services.net/ns/rm#">
  <oslc:ResponseInfo rdf:about="http://prototype.v2c2.at:8081/RMProvider/
      services/requirements">
    <rdfs:member>
      <oslc_rm:Requirement rdf:about="http://prototype:8081/RMProvider/
    services/requirements/Req2">
        <oslc:instanceShape rdf:resource="http://prototype:8081/RMProvider
            /services/resourceShapes/requirement/requirements"/>
        <dcterms:created rdf:datatype="http://www.w3.org/2001/
            XMLSchema#dateTime">2016−07−23T07:14:46.99Z</dcterms:
            created>
        <dcterms:modified rdf:datatype="http://www.w3.org/2001/
            XMLSchema#dateTime">2016−07−23T07:14:46.99Z</dcterms:
            modified>
        <oslc:serviceProvider rdf:resource="http://prototype:8081/
            RMProvider/services/serviceProviders/1"/>
        <dcterms:identifier>Req2</dcterms:identifier>
        <dcterms:title rdf:parseType="Literal">Battery Requirement 3</
    dcterms:title>
```

```
        <dcterms:description rdf:parseType="Literal">The SOC shall be between 20-80%.</dcterms:description>
      </oslc_rm:Requirement>
    </rdfs:member>
    <rdfs:member>
      <oslc_rm:Requirement rdf:about="http://prototype:8081/RMProvider/services/requirements/Req1">
        <oslc:instanceShape rdf:resource="http://prototype:8081/RMProvider/services/resourceShapes/requirement/requirements"/>
        <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2016-07-23T07:14:46.99Z</dcterms:created>
        <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2016-07-23T07:14:46.99Z</dcterms:modified>
        <oslc:serviceProvider rdf:resource="http://prototype:8081/RMProvider/services/serviceProviders/1"/>
        <dcterms:identifier>Req1</dcterms:identifier>
        <dcterms:title rdf:parseType="Literal">Battery Requirement 2</dcterms:title>
        <dcterms:description rdf:parseType="Literal">The battery termperature shall be between 10 and 30 deg C.</dcterms:description>
      </oslc_rm:Requirement>
    </rdfs:member>
    <rdfs:member>
      <oslc_rm:Requirement rdf:about="http://prototype:8081/RMProvider/services/requirements/Req0">
        <oslc:instanceShape rdf:resource="http://prototype:8081/RMProvider/services/resourceShapes/requirement/requirements"/>
        <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2016-07-23T07:14:46.99Z</dcterms:created>
        <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2016-07-23T07:14:46.99Z</dcterms:modified>
        <oslc:serviceProvider rdf:resource="http://prototype:8081/RMProvider/services/serviceProviders/1"/>
        <dcterms:identifier>Req0</dcterms:identifier>
        <dcterms:title rdf:parseType="Literal">Battery Requirement 1</
    dcterms:title>
        <dcterms:description rdf:parseType="Literal">When accelerating
            for a period of 30 sec using boost mode, the battery
            temperature increase should not exceed a delta temperature
            of 20 deg C.</dcterms:description>
      </oslc_rm:Requirement>
    </rdfs:member>
    <oslc:totalCount rdf:datatype="http://www.w3.org/2001/XMLSchema#
        int">3</oslc:totalCount>
  </oslc:ResponseInfo>
</rdf:RDF>
```

Listing A.3: RM Tool Query Capability

OSLC Requirement

HTTP GET http://prototype:8081/RMProvider/services/requirements/Req0

```
<rdf:RDF
    xmlns:oslc="http://open-services.net/ns/core#"
    xmlns:oslc_rm="http://open-services.net/ns/rm#">
  <rdf:Description rdf:about="http://prototype:8081/RMProvider/services/
      requirements/Req0">
    <oslc:instanceShape rdf:resource="http://prototype:8081/RMProvider/
        services/resourceShapes/requirement/requirements"/>
    <dcterms:created rdf:datatype="http://www.w3.org/2001/XMLSchema#
        dateTime">2016-07-23T07:14:46.99Z</dcterms:created>
    <dcterms:modified rdf:datatype="http://www.w3.org/2001/XMLSchema#
        dateTime">2016-07-23T07:14:46.99Z</dcterms:modified>
    <oslc:serviceProvider rdf:resource="http://prototype:8081/RMProvider/
        services/serviceProviders/1"/>
    <dcterms:identifier>Req0</dcterms:identifier>
    <dcterms:title rdf:parseType="Literal">Battery Requirement 1</
    dcterms:title>
    <dcterms:description rdf:parseType="Literal">When accelerating for
        a period of 30 sec using boost mode, the battery temperature
        increase should not exceed a delta temperature of 20 deg C.</
        dcterms:description>
    <rdf:type rdf:resource="http://open-services.net/ns/rm#Requirement"/>
  </rdf:Description>
</rdf:RDF>
```

Listing A.4: RM Tool Requirement

A.2. TRS Messages

Tracked Resource Set (TRS)

HTTP GET http://prototype:8081/RMProvider/services/trs

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:trs="http://open-services.net/ns/core/trs#">
  <trs:TrackedResourceSet rdf:about="http://prototype:8081/RMProvider/
      services/trs">
    <trs:base rdf:resource="http://prototype:8081/RMProvider/services/trs/
        base"/>
    <trs:changeLog>
      <trs:ChangeLog>
        <trs:previous rdf:resource="http://prototype:8081/RMProvider/
    services/trs/changeLog/2"/>
        <trs:change>
          <trs:Creation rdf:about="http://prototype:8081/RMProvider/
    services/trs:2016-07-23T09:05:17Z:3">
            <trs:changed rdf:resource="http://prototype:8081/RMProvider/
                services/requirements/Req4"/>
            <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#
                int">3</trs:order>
          </trs:Creation>
        </trs:change>
        <trs:change>
          <trs:Creation rdf:about="http://prototype:8081/RMProvider/
    services/trs:2016-07-23T09:05:17Z:4">
            <trs:changed rdf:resource="http://prototype:8081/RMProvider/
                services/requirements/Req3"/>
            <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#
                int">4</trs:order>
          </trs:Creation>
        </trs:change>
        <trs:change>
          <trs:Creation rdf:about="http://prototype:8081/RMProvider/
    services/trs:2016−07−23T09:05:17Z:5">
            <trs:changed rdf:resource="http://prototype:8081/RMProvider/
                services/requirements/Req2"/>
            <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#
                int">5</trs:order>
          </trs:Creation>
        </trs:change>
      </trs:ChangeLog>
    </trs:changeLog>
  </trs:TrackedResourceSet>
</rdf:RDF>
```

Listing A.5: Tracked Resource Set (TRS)

TRS Base

HTTP GET http://prototype:8081/RMProvider/services/base

```
<?xml version="1.0" encoding="UTF−8"?>
<rdf:RDF>
  <ldp:Page rdf:about="http://prototype:8081/RMProvider/services/trs/base1"
    >
    <ldp:pageOf>
      <ldp:Container rdf:about="http://prototype:8081/RMProvider/services/
    trs/base">
        <trs:cutoffEvent rdf:resource="http://prototype:8081/RMProvider/
            services/trs:2016−07−23T09:05:17Z:5"/>
        <rdfs:member rdf:resource="http://prototype:8081/RMProvider/
            services/requirements/Req4"/>
        <rdfs:member rdf:resource="http://prototype:8081/RMProvider/
            services/requirements/Req3"/>
        <rdfs:member rdf:resource="http://prototype:8081/RMProvider/
            services/requirements/Req2"/>
      </ldp:Container>
    </ldp:pageOf>
    <ldp:nextPage rdf:resource="http://prototype:8081/RMProvider/services/
        trs/base/2"/>
  </ldp:Page>
</rdf:RDF>
```

Listing A.6: TRS Base Resource

TRS Change Log

HTTP GET http://prototype:8081/RMProvider/services/trs/changeLog

```
<?xml version="1.0" encoding="UTF−8"?>
<rdf:RDF
    xmlns:oslc="http://open−services.net/ns/core#">
  <trs:ChangeLog rdf:about="http://prototype:8081/RMProvider/services/trs/
      changeLog">
    <trs:previous rdf:resource="http://prototype:8081/RMProvider/services/
        trs/changeLog/2"/>
    <trs:change>
      <trs:Creation rdf:about="http://prototype:8081/RMProvider/services/
    trs:2016−07−23T09:05:17Z:3">
        <trs:changed rdf:resource="http://prototype:8081/RMProvider/
            services/requirements/Req4"/>
        <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >3</trs:order>
      </trs:Creation>
    </trs:change>
    <trs:change>
      <trs:Creation rdf:about="http://prototype:8081/RMProvider/services/
    trs:2016−07−23T09:05:17Z:4">
        <trs:changed rdf:resource="http://prototype:8081/RMProvider/
            services/requirements/Req3"/>
        <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >4</trs:order>
      </trs:Creation>
    </trs:change>
    <trs:change>
      <trs:Creation rdf:about="http://prototype:8081/RMProvider/services/
    trs:2016−07−23T09:05:17Z:5">
```

```
            <trs:changed rdf:resource="http://prototype:8081/RMProvider/
                services/requirements/Req2"/>
            <trs:order rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
                >5</trs:order>
          </trs:Creation>
        </trs:change>
      </trs:ChangeLog>
    </rdf:RDF>
```

Listing A.7: TRS Change Log

Bibliography

[AC01] Ryan Ausanka-Crues. Methods for access control: advances and limitations. *Harvey Mudd College*, 301, 2001.

[AHL$^+$14] Bernhard K Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Rupert Schlick, Didier Simoneau, and Stefan Tiran. Integration of Requirements Engineering and Test-Case Generation via OSLC. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 117–126. IEEE, 2014.

[ANN03] Nadarajah Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *International Workshop on Security Protocols*, pages 28–41. Springer, 2003.

[BBDLM14] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[BBG07] Rafae Bhatti, Elisa Bertino, and Arif Ghafoor. An Integrated Approach to Federated Identity and Privilege Management in Open Systems. *Communications of the ACM*, 50(2):81–87, 2007.

[Bih96] Eli Biham. How to Forge DES-Encrypted Messages in 2ˆ 28 Steps. *Technion Computer Science Department Technical Report CS0884*, 1996.

[BSJ15] John Bradley, Nat Sakimura, and Michael Jones. JSON Web Token (JWT). RFC 1654, RFC Editor, 2015. URL: `http://www.rfc-editor.org/rfc/rfc1654.txt`.

[BT11] Elisa Bertino and Kenji Takahashi. *Identity Management: Concepts, Technologies, and Systems*. Artech House, 2011.

[bug16] Bugzilla. `https://www.bugzilla.org/`, 2016. Accessed: 2016-11-30.

[Cas02] Marco Casole. WLAN Security-Status, Problems and Perspective. In *Proceedings of European Wireless*, 2002.

[CH13] Yulia Cherdantseva and Jeremy Hilton. A Reference Model of Information Assurance & Security. *2013 International Conference on Availability, Reliability and Security*, pages 546–555, 2013.

[com13] Common Criteria (2013). `http://www.commoncriteriaportal.org/`, 2013. Accessed: 2016-11-30.

[Cornd] International Business Machines Corporation. IBM Jazz. `https://jazz.net/`, n.d. Accessed: 2016-11-30.

[CSF$^+$08] D. Cooper, S. Santesson, S. Farell, S. Boeyen, Russell Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and CRL profile. RFC 5280, RFC Editor, 2008. URL: `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[Dav11] Joshua Davies. *Implementing SSL/TLS using cryptography and PKI*. John Wiley and Sons, 2011.

[DR08] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, 2008. URL: `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[DR11] Thai Duong and Juliano Rizzo. Here come the XOR ninjas. *Unpublished manuscript*, 320, 2011.

[ecl16] Eclipse Lyo. `http://www.eclipse.org/lyo/`, 2016. Accessed: 2016-11-30.

[ECPB12] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. Prentice Hall Press, 2012.

[FHBH+99a] John Franks, P Hallam-Baker, J Hostetler, S Lawrence, P Leach, Ari Luotonen, and L Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, RFC Editor, 1999. URL: `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[FHBH+99b] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, 1999. URL: `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[FK92] David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. *15th National Computer Security Conference*, pages 554–563, 1992.

[Fou16a] Apache Software Foundation. Apache Oltu. `https://oltu.apache.org/`, 2016. Accessed: 2016-11-30.

[Fou16b] Eclipse Foundation. Eclipse RDF4J. `http://rdf4j.org/`, 2016. Accessed: 2016-11-30.

[Fou17] Eclipse Foundation. SWT: The Standard Widget Toolkit. `https://www.eclipse.org/swt/`, 2017. Accessed: 2017-01-15.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[GWMH15] Kelly Grizzle, Erik Wahlstroem, Chuck Mortimore, and Phil Hunt. System for Cross-domain Identity Management: Core Schema. RFC 6749, RFC Editor, 2015. URL: `http://www.rfc-editor.org/rfc/rfc7643.txt`.

[Ham09] Eran Hammer. Explaining the OAuth Session Fixation Attack. `https://hueniverse.com/2009/04/23/explaining-the-oauth-session-fixation-attack`, 2009. Accessed: 2016-10-16.

[Ham12] Eran Hammer. OAuth 2.0 and the Road to Hell. `https://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell`, 2012. Accessed: 2016-10-21.

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, 2012. URL: `http://www.rfc-editor.org/rfc/rfc6749.txt`.

[HLOS06] Shawn Hernan, Scott Lambert, Tomasz Ostwald, and Adam Shostack. Threat Modeling - Uncover Security Design Flaws Using The STRIDE Approach. *MSDN Magazine-Louisville*, pages 68–75, 2006.

[HMMC15] Thomas Hardjono, Eve Maler, Maciej Machulak, and Domenico Catalano. User-Managed Access (UMA) Profile of OAuth 2.0. *Kantara Initiative*, 2015.

[HMW+12] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 413–428, Bellevue, WA, 2012. USENIX. URL: `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang`.

[ISO09] ISO/IEC. Information technology – Security techniques – Information security management systems – Overview and vocabulary, 2009.

[ISO11] ISO/IEC. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[JH12] Michael B. Jones and Dick Hardt. The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, RFC Editor, 2012. URL: `http://www.rfc-editor.org/rfc/rfc6750.txt`.

[JZI10] Maziar Janbeglou, Mazdak Zamani, and Suhaimi Ibrahim. Redirecting network traffic toward a fake DNS server on a LAN. In

3rd IEEE International Conference on Computer Science and Information Technology, pages 429–433, 2010.

[Kha12] Abdul Raouf Khan. Access control in cloud computing environment. *ARPN Journal of Engineering and Applied Sciences*, 7(5):613–615, 2012.

[KMZ02] Steve Kremer, Olivier Markowitch, and Jianying Zhou. An intensive survey of fair non-repudiation protocols. *Computer Communications*, 25(17):1606 – 1621, 2002. URL: `http://www.sciencedirect.com/science/article/pii/S014036640200049X`, `doi:http://dx.doi.org/10.1016/S0140-3664(02)00049-X`.

[LMH13] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. OAuth 2.0 Threat Model and Security Considerations. RFC 6819, RFC Editor, 2013. URL: `http://www.rfc-editor.org/rfc/rfc6819.txt`.

[LPL$^+$03] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First Experiences Using XACML for Access Control in Distributed Systems. In *Proceedings of the 2003 ACM Workshop on XML Security*, XMLSEC '03, pages 25–37. ACM, 2003. URL: `http://doi.acm.org/10.1145/968559.968563`, `doi:10.1145/968559.968563`.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. *https://www.openssl.org/ bodo/ssl-poodle.pdf*, 2014.

[Men07] Falko Menge. Enterprise service bus. In *Free and open source software conference*, volume 2, pages 1–6, 2007.

[MKL09] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.* "O'Reilly Media, Inc.", 2009.

[MLHW15] Nadja Marko, Andrea Leitner, Beate Herbst, and Alfred Wallner. Combining Xtext and OSLC for Integrated Model-Based Requirements Engineering. In *Software Engineering and Advanced*

Applications (SEAA), 2015 41st Euromicro Conference on, pages 143–150. IEEE, 2015.

[MMM+04] Frank Manola, Eric Miller, Brian McBride, et al. RDF Primer. *W3C recommendation*, 10(1-107):6, 2004.

[MNN14] Abhishek Majumder, Suyel Namasudra, and Samir Nath. Taxonomy and classification of access control models for cloud environments. *Continued Rise of the Cloud*, pages 23–55, 2014.

[MS13] Christopher Meyer and Jörg Schwenk. Sok: Lessons learned from ssl/tls attacks. In *International Workshop on Information Security Applications*, pages 189–209. Springer, 2013.

[NB13] Moustafa Noureddine and Rabih Bashroush. An authentication model towards cloud federation in the enterprise. *Journal of Systems and Software*, 86(9):2269–2275, 2013.

[net10] net.oauth - OAuth 1.0 Revision A Library. `http://repo1.maven.org/maven2/net/oauth/`, 2010. Accessed: 2016-08-14.

[OAS13] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard, 2013.

[OAS14] OASIS. XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile Version 1.0. OASIS Standard, 2014.

[OAu09] OAuth Core Workgroup. OAuth Core 1.0 Revision A. `http://oauth.net/core/1.0a/`, 2009. Accessed: 2016-04-10.

[OHB06] Rolf Oppliger, Ralf Hauser, and David Basin. SSL/TLS session-aware user authentication–Or how to effectively thwart the man-in-the-middle. *Computer Communications*, 29(12):2238–2246, 2006.

[oO05] Security Services Technical Committee of OASIS. SAML V2.0 Standard. OASIS Standard, 2005. URL: `https://wiki.oasis-open.org/security/FrontPage`.

[osl12] Open Services for Lifecycle Collaboration Requirements Management Specification Version 2.0. `http://open-services.net/bin/view/Main/RmSpecificationV2`, 2012. Accessed: 2017-01-07.

[osl13] Open Services for Lifecycle Collaboration Core Specification Version 2.0. `http://open-services.net/bin/view/Main/OslcCoreSpecification`, 2013. Accessed: 2017-01-03.

[osl15a] Open Services for Lifecycle Collaboration Indexable Linked Data Provider Specification Version 2.0. `http://open-services.net/wiki/core/IndexableLinkedDataProvider-2.0/`, 2015. Accessed: 2016-11-30.

[osl15b] Open Services for Lifecycle Collaboration Tracked Resource Set Specification Version 2.0. `http://open-services.net/wiki/core/TrackedResourceSet-2.0`, 2015. Accessed: 2016-11-30.

[osl16] Open Services for Lifecycle Collaboration (OSLC). `http://open-services.net/`, 2016. Accessed: 2017-01-03.

[OWA16] OWASP. OWASP Top Ten Project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2016. Accessed: 2016-10-30.

[PC09] Siani Pearson and Andrew Charlesworth. Accountability as a Way Forward for Privacy Protection in the Cloud. In *Cloud computing*, pages 131–144. Springer, 2009.

[PMC14] Tim Polk, Kerry McKay, and Santosh Chokhani. Guidelines for the selection, configuration, and use of transport layer security (TLS) implementations. *NIST Special Publication*, 800:52, 2014.

[PSH08] Eric Prud'Hommeaux, Andy Seaborne, and Steve Harris. SPARQL query language for RDF. *W3C recommendation*, 2008.

[RBBJ10] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: A Study of Clickjacking Vulnerabilities at Popular Sites. *IEEE Oakland Web*, 2:6, 2010.

[Res00] Eric Rescorla. HTTP Over TLS. RFC 2818, RFC Editor, 2000. URL: `http://www.rfc-editor.org/rfc/rfc2818.txt`.

[Riv92] Ronald Rivest. The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor, 1992. URL: `http://www.rfc-editor.org/rfc/rfc1321.txt`.

[Riv05] Ronald Rivest. [Python-Dev] hashlib - faster md5/sha, adds sha256/512 support. `https://mail.python.org/pipermail/python-dev/2005-December/058850.html`, 2005. Accessed: 2016-10-20.

[RK03] Eric Rescorla and Brian and Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552, RFC Editor, 2003. URL: `http://www.rfc-editor.org/rfc/rfc3552.txt`.

[Rou12] Derrick Rountree. *Federated Identity Primer*. Syngress, 2012.

[SBJ$^+$14] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. OpenID Connect Core 1.0. *The OpenID Foundation*, 2014.

[Sir14] Prabath Siriwardena. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Apress, 2014.

[SKV$^+$16] Dimitris E Simos, Kristoffer Kleine, Artemios G Voyiatzis, Rick Kuhn, and Raghu Kacker. TLS Cipher Suites Recommendations: A Combinatorial Coverage Measurement Approach. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 69–73. IEEE, 2016.

[Suz16] Bojan Suzic. Securing Integration of Cloud Services in Cross-domain Distributed Environments. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 398–405. ACM, 2016. URL: `http://doi.acm.org/10.1145/2851613.2851622`, `doi:10.1145/2851613.2851622`.

[Was90] Anthony I Wasserman. Tool integration in software engineering environments. In *Software Engineering Environments*, pages 137–149. Springer, 1990.

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive*, 2004:199, 2004.

[WM11] Michael E Whitman and Herbert J Mattord. *Principles of information security.* Cengage Learning, 2011.

[YM13] Feng Yang and Sathiamoorthy Manoharan. A security analysis of the OAuth protocol. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 271–276. IEEE, 2013.

[ZZ14] Thomas Zefferer and Bernd Zwattendorfer. An Implementation-independent Evaluation Model for Server-based Signature Solutions. *10th International Conference on Web Information Systems and Technologies (WEBIST)*, pages 302–309, 2014.