Reference monitors for security and interoperability in OAuth 2.0*

Ronan-Alexandre Cherrueau[1], Rémi Douence[1], Jean-Claude Royer[1], Mario Südholt[1], Anderson Santana de Oliveira[2], Yves Roudier[3], and Matteo Dell'Amico[3]

[1] École des Mines de Nantes, Nantes, France
[2] SAP Applied Research, Mougins, France
[3] EURECOM, Sophia Antipolis, France

Abstract. OAuth 2.0 is a recent IETF standard devoted to providing authorization to clients requiring access to specific resources over HTTP. It was recently adopted by major internet players like Google, Facebook, and Microsoft. It has been pointed out that this framework is potentially subject to security issues, as well as difficulties concerning the interoperability between protocol participants and application evolution. As we show in this paper, there are indeed multiple reasons that make this protocol hard to implement and impede interoperability in the presence of different kinds of client. Our main contribution consists in a framework that harnesses a type-based policy language and aspect-based support for protocol adaptation through flexible reference monitors in order to handle security, interoperability and evolution issues of OAuth 2.0. We apply our framework in the context of three scenarios that make explicit variations in the protocol and show how to handle those issues.

1 Introduction

Web services and applications are implemented more and more frequently using open standards for security goals such as WS-policy for SOAP-based services, and, more commonly as part of RESTful APIs, OpenID for authentication as well as OAuth for authorization. OAuth has gained a lot of interest, its 2.0 version recently becoming an IETF standard. All major internet players (Google, Facebook, Microsoft, among others) have already released API's to allow resource access delegation in web applications using this standard.

Although the specifications of the standard are sufficiently clear, developers often have difficulties to correctly implement all of its features. There

* This work has been partially supported by the CESSA ANR project (ANR 09-SEGI-002-01, http://cessa.gforge.inria.fr) and the A4Cloud project (FP7 317550, http://www.a4cloud.eu/).

are frequently subject to general problems concerning security and interoperability. For example, the design of OAuth 2.0 has put forward simplicity instead of security when choosing to support bearer tokens, which do not require to prove the possession of a cryptographic key. Token confidentiality relies then on storage and transport security (SSL/TLS); therefore, all resources mediated via OAuth 2.0 would be exposed if the transport layer security breaks (in the following, we will use simply "OAuth" instead of OAuth 2.0).

Another problem developers face when using OAuth is to actually produce interoperable implementations. The OAuth standard is not simply an authentication and delegation protocol, but an "authorization framework," whose design was heavily influenced by enterprise use cases. In order to support those use cases, the standard allows for extensibility and defines several components as optional. The standard also specifies several important features only partially, such as client registration, authorization server capabilities, and endpoint discovery, all features that are fundamental to automate service compositions in real implementations.

In this paper we provide a framework that integrates three main features in order to enable programmers to handle such security and interoperability issues, as well as related evolution scenarios: *i)* an abstract and typed language for the high-level definition of security policies over service interactions, *ii)* the HiPoLDS [10] model for flexible reference monitors, and *iii)* aspect-oriented programming techniques for the manipulation of service implementations. More concretely, we provide four corresponding contributions. First, we show how to use a type system with explicit channel types and service subtypes in order to provide correctness guarantees over service compositions and to improve interoperability of the OAuth framework. Second, we harness the high level abstract policy language HiPoLDS for the definition of flexible reference monitors that help enforcing policies on the message level. Third, we leverage a set of aspect-oriented secure software development techniques to manage the evolution of service security capabilities and decouple them from the underlying service implementation; overall, we thus increase the dependability of OAuth deployments. Finally, we apply these techniques in the context of three realistic scenarios that exhibit security, interoperability and evolution issues of the OAuth standard.

This paper is structured as follows. Section 2 introduces the OAuth framework and some of its issues. Section 3 is dedicated to the description of the typed service language and the techniques for service manipulation

we use. An application to OAuth in the context of three scenarios is described in Sec. 4. We finish with related work in Sec. 5 and a conclusion.

2 The OAuth 2.0 Authorization Framework

OAuth is an IETF standard devoted to providing authorization to clients requiring access to specific resources over HTTP. The standard was issued as RFC 6749 [22] in October 2012 and is not compatible with the first version. Several web application providers are currently using this framework, among them: Google, GitHub, Windows Live, and Facebook. OAuth defines several protocols for resource owners to grant third-party access to their resources without exposing their passwords to resource users.

2.1 The Authorization Code Flow Case Study

We will concentrate our study on a central part of the protocol, the *Authorization Code Flow* (or ACF), which is described in Section 4.1 of the standard. The general architecture is depicted in Figure 1. This protocol assumes several parties with different roles. The Resource Owner (RO) is an entity (either a human being, the end-user, or some software he uses) that grants access to some protected resources. A client (C) is a third-party application requesting the use of resources owned by the resource owner. The Authorization Server (AS) is a software application dedicated to checking client rights to access protected resources and delivering related access tokens. The User Agent (UA) is a software application which mediates communications between the client, the resource owner, and the authorization server. The authorization server has two HTTP endpoints: The authorization request (arep) and the token request (trep), while the client requires only one endpoint (crep). There are two types of clients: confidential or public depending if they are capable (or not) of maintaining the confidentiality of their credentials (password, identity, authorization code, token, ...). To get an access token the client C interacts with the authorization server in order to first get an authorization code. This authorization code is delivered by the AS to the client on the behalf of the resource owner. The client and the resource owner do not directly interact in this setting. The protocol assumes that the RO and the clients are registered to the AS. At registration time the confidential client gives an identifier and a URI. For a public client, the authentication method is optional and depends from the AS requirements.

The communication steps of the Authorization Code Flow, depicted in Figure 1, are as follows:

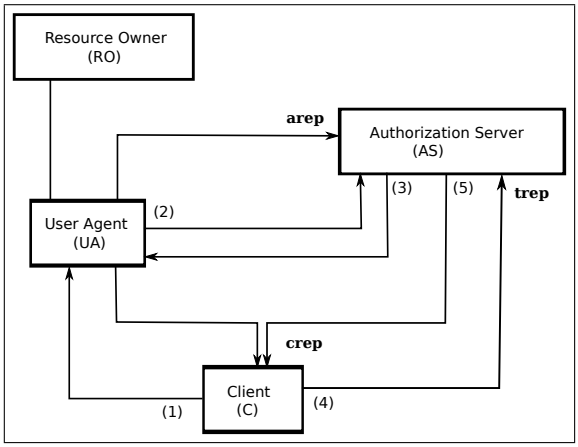

Fig. 1. The Authorization Code Flow (ACF)

1. C initiates a request and directs UA to AS. C includes its identifier, a state, and optionally a scope and a URI in the message.
2. RO is authenticated via its user agent UA. In this step RO grants or denies access to C.
3. AS replies to C (via UA) with either an authorization code or an error code.
4. C requests an access token from AS. C uses the token endpoint and includes its authorization code, and a URI to redirect the reply.
5. AS authenticates C and checks that the authorization code was previously delivered to C. Authentication is mandatory for confidential clients or if an authentication scheme has been previously established with a public client. AS also checks that if a URI was provided, it is the same as the URI provided when C requested the authorization code. If all these controls are valid, AS sends a token to the redirection URI (and optionally a refresh token).

The OAuth framework specifies further details about the authorization code and token in RFC 6750 [23]. The recommended time life for an authorization code is 10 minutes and it must not be used more than once. Access tokens are credentials used to access the protected resources stored on a resource server and they have a specific scope and a duration limit.

2.2 Interoperability, Security and Evolution Issues

We now present the relevant problems faced by OAuth implementations.

Interoperability. The OAuth standard has been criticized[4] for its likelihood in producing non-interoperable implementations. There are multiple sources for this problem. A large number of components are optional, for example; tokens may assume the "bearer" or "MAC" formats according to the standard, or yet SAML assertions may be used [21]. Furthermore, several components are only partially defined in the standard; this applies, in particular, for the client registration process, server authorization capabilities, and endpoint discovery mechanism. Generally, developers of OAuth client application are interested in creating services that are as flexible as possible in order to be able to access data from multiple resource servers. Because of the interoperability issues, this requires the handling of a large number of distinct settings for each different authorization server, raising maintainability and reusability difficulties.

Evolution. As OAuth is a web authorization framework, its adoption in diverse enterprise scenarios is to be expected. Existing implementations need to be modified in order to cover requirements coming from the enterprise world: resource owners, for example, are unlikely to be individuals but rather organizations. Therefore we envisage in this paper a scenario where authorization needs to be obtained from a user on behalf of its organization.

Security. Several security problems of OAuth are known and the specification warns about a number of potential security issues (Sec. 10 of [22]). Furthermore, threats related to injection attacks and the insufficient protection of credentials have also been investigated [12,4].

3 A Typed Framework for Policy Enforcement

In this section we introduce the framework we leverage to solve the security, evolution and interoperability problems of OAuth. Our solution relies on a typed policy language for service interactions and two main concepts for service manipulations: aspect-oriented programming and reference monitors for policy enforcement.

3.1 Typed Service Interactions

We propose to use a rich type system for service interactions which is sound even in presence of attackers [2]. This type system is defined using

[4] See, e.g., Hammer: `hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell`

so-called semantic typing [5], it supports negation, intersection and union types which are convenient in a query or declarative context. Adding subtyping is important for two main reasons: *i)* it extends dynamic channel discovery since required services may be provided by more specific ones and *ii)* it improves interoperability, a client can connect to various compatible services. Since we can discover new channels at runtime, type inference is done at message reception time. Type inference checks that the message is well-formed and computes the types of the discovered channels in the messages.

Concretely, our type system provides the following types: Classic basic types (like `String, Integer, ...`), structured types as labeled type list (`"label"[Type],Type`), record types (`{"label":Type; ...}`) and type for channels (or URIs) that are denoted `<Type>`. We have also negation types (`NOT Type`), union types (`Type + Type`), and intersection types (`Type & Type`). Furthermore we type provided endpoints (channel, URI) as well as required ones. For the definition of the type system, see [2].

This type system has the following benefits: *i)* it makes explicit a contract that has to be obeyed by servers and clients, *ii)* it is subject to verification and avoids some ill-formed messages that result from errors or code injection attacks, *iii)* it provides powerful and declarative means to define properties of data, channels and parties in communications, *iv)* it supports subtyping which is convenient for more flexible discovery and interoperability. Currently the type system and its machinery has been implemented as a Java library. Work on the integration of the type system in Apache's service framework CXF is on-going.

3.2 Security Domains and Policies

In order to extend OAuth with security policies about resource access, we are using the HiPoLDS language we defined in [10]. Security policies in HiPoLDS rely on the description of the information flows between so-called policy domains. Those domains can capture both component and protocol entities. Policies are expressed using rules that match with the content or with specific properties of the information flow. In particular, HiPoLDS describes patterns in the flow based on the notion of information tag, a construct of the policy language used to annotate the message with security metadata. Some tags can relate to the content of a message payload, at different levels in a protocol stack, like the IP address, some field value at a given offset in the payload, or an encrypted blob in some other part of the message; alternately, other tags refer to more structural component or protocol concepts. In particular, the type system described

above can be seen as an example of the latter category: types can be introduced into HiPoLDS rules by annotating the message with a specific information tag. Payload related tags can also be identified through message annotation at the type inference phase. Section 4.2 illustrates both situations.

3.3 Monitors and Aspects

The implementation of both the type system and of policy enforcement at the protocol level can be done with reference monitors [18,8]. Reference monitors represent a flexible solution to evolve existing applications without modifying their code. They act as wrappers around agents and intercept incoming and outgoing messages. Many actions can be associated with messages: control, remove, modify, resend, etc. This is for instance a good way to add extra control on messages to avoid some attacks. In our case, we use monitors, implemented using the HiPoLDS rules, to secure the storage of credentials and to oblige agents to use SSL/TLS connections as advocated by the OAuth standard.

Sometimes we need more intrusive actions to modify the internal code of agents. In this case we propose to use an aspect-oriented approach to complement the monitors. To this end, we have defined a (new) aspect system for Apache's CXF service framework, see [3] for a publicly available implementation. This aspect system enables programmers to statically or dynamically modify service compositions, interceptor definitions and Java-based implementations of CXF services. The events that trigger modifications are defined in terms of finite-state based sequences of service invocations, interceptor calls or features of the service implementation. Once such events are identified, new Java code may be injected or used to replace existing code.

4 Application to OAuth

We now demonstrate how our techniques can overcome the issues impacting OAuth introduced in Section 2.2. To this end we consider a workflow from the banking domain, see Fig. 2, as part of which a bank and an insurance company together provide services to private customers.

Alice is a customer of the bank where she has contracted a loan. The bank proposes Alice to use third party services to acquire an insurance concerning her loan and buy a share portfolio. For that, Alice uses her web browser to open the web service from the insurance company, which

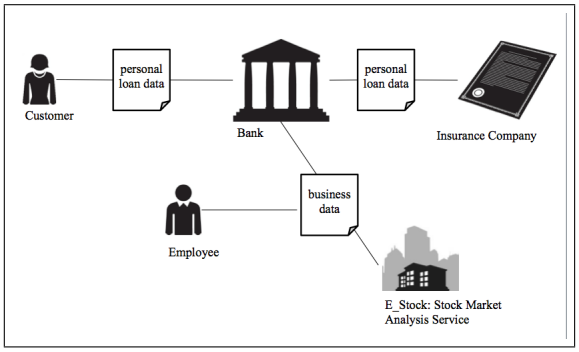

Fig. 2. Enterprise usage of OAuth

requests access to her loan data stored at the bank. In OAuth terms, Alice is thus the resource owner with her web browser as user agent, the insurance company plays the role of the client, and the bank acts as a resource and an authorization server. Note that Alice is not bound to use only the service of the insurance company (as she is the only responsible for her personal data); she can therefore choose to use any other client registered to the bank's authorization server.

In the following we consider three OAuth-related extension scenarios and show how our framework solves the interoperability and evolution issues raised by these scenarios.

4.1 Type-based Definition of OAuth-conform Interactions

Scenario 1. As a first interoperability scenario we assume that the client, the insurance company, was built to work with an existing AS server, e.g., provided by its headquarters, which does not issue refresh tokens. Provided that the same client will request access to resources held in the resource server, the problem now is to equip or modify the client implementation such that it will also be able to use refresh tokens, as imposed by the AS from Alice's bank.

Another token-related interoperability issue consists in different types of access tokens. The OAuth standard allows for bearer and MAC tokens. For instance, the client was built to use bearer tokens, whereas the Bank server requires the MAC token type.

Solution. In order to handle incompatibilities between stakeholders we have to be able to define suitable channel types for the endpoints for

OAuth-related interactions and then provide suitable types for the different token types as discussed in scenario 1 above.

The client endpoint (called redirection URI of the client and noted `crep`) should receive rich information from the AS, and its type can be defined as

```
crep = < ( {"grant":AuthCode; "state":State}
           + Token + DenyError ) & Secure >
```

On this endpoint the client can receive an authentication code with a state or a token or an error, this is a union type noted `+`. It further specifies that the client should receive secure information with an intersection type (noted `&`). Each type should be as complex as needed, for instance specifying the various cases of errors or the fact that the authorization code and the token have a time duration. Agents are responsible to implement the types and to use values according to their types, types explicit a contract the interacting parties must observe.

The authorization endpoint provided by the bank has type:

```
arep = < ( { "id":Credents ; "state":State }
           ⊕ Scope ⊕ C.crep ) & Secure >
```

Mandatory information (client identifier, secret and state) are collected in a record type, while optional information (scope and client redirection URI) is typed using the $\oplus$ operator. This is a syntactic sugar for a combination of record and union types defining optional information type. Note that we found the provided `C.crep` type from `C` in `arep` since the client has the option to send its proper URI to the AS.

In the first scenario we need different kinds of token, which can be represented by the subtype hierarchy depicted in Figure 3.

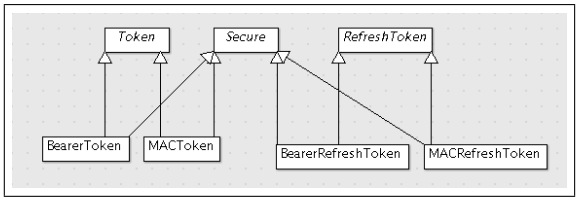

Fig. 3. A UML like Hierarchy of Tokens

The hierarchy uses unrelated types for "real" tokens and refresh tokens because the latter are not used for resource access but for token management. The client URI should be connected to several servers and with

two required endpoints (`AS.crep` (3) and `AS.crep` (5)). Component typing rules imply that these required endpoints should be supertypes of the `C.crep` channel type. For instance, `<(Token + DenyError) & Secure>` or `<MACToken>` are such supertypes (this can be easily shown using the rules in appendix A Table 1). The type-checking ensures this control and excludes dynamic type errors.

In scenario 1, the client should receive either tokens from the insurance AS, or tokens and refresh tokens from the bank AS. All of this token information could be secured using either bearer or MAC kinds of token. We have then to change the `crep` type. We can use for instance, one endpoint with the following type: `crep = <(AuthCode + Token + {"token":Token; "refresh":RefreshToken} + DenyError) & Secure>`.

To evolve the client, we encapsulate it into a reference monitor which manages types, incoming and outgoing messages. To handle the different interoperability situations, the monitor for the client could either have a unique general URI as above or several dedicated URIs connected to each server endpoint. Defining only one endpoint for the client is better at least from a coupling point of view. Decreasing component coupling increases the endpoint type complexity and this requires a powerful type system as the one proposed here. In this case, the adaptation code checks the dynamic type of the received values in messages and triggers additional codes. This is the place where HiPoLDS rules or aspects can be used as shown below.

4.2 Extending the OAuth Framework Using a Policy

Scenario 2. The OAuth framework describes the overall protocol to grant clients access tokens to the resource server using the Authorization Code Flow. We consider here a scenario in which we need to extend the protocol in order to handle additional security strategies.

Consider that David, a bank employee, needs to analyze the profitability of the fund he manages. In order to do so, he uses an external service from the stock market analysis company E_Stock to evaluate the fund portfolio that contains no personally identifiable information about bank customers. Clearly, E_Stock is acting as a client with respect to the Bank, which still plays the role of a resource server and owner.

The difference to OAuth's standard protocol is here that David cannot be considered as the only resource owner since the actual data owner is the bank. Furthermore, we consider another requirement: banks today typically need to ensure additional accountability guarantees with respect to their employees' behavior. For instance, David should not be able to

delegate access to arbitrary external services. He should also not be able to delegate access to stock managed by the bank outside of his fund portfolio.

Solution. The OAuth framework thus has to be extended with the enforcement of a mandatory security policy defined by the bank with respect to its employees' actions. The User Agent's authentication is therefore itself subject to the granting of an authorization by the bank.

The implementation of this mandatory access control on top of OAuth depends essentially on the entity that runs the AS. If it is the bank, then the AS provides a perfect point of enforcement; otherwise, if the AS is managed by a third party, the bank will need to intercept messages between the client and the AS. In the latter case, an aspect based implementation of the reference monitor is necessary as network traffic is likely encrypted (the use of an SSL/TLS secure session being typically recommended in the OAuth framework), whereas in the former case, the reference monitor can be directly introduced by the bank after decryption.

In this scenario, a HiPoLDS reference monitor at the bank would make sure that David only authorizes reading data about stock from his own portfolio, and that the client is an acceptable third party, as identified among a set of authorized services. The following HiPoLDS rule expresses these constraints, by dropping messages not conforming to the policy:

```
m:arep, m.scope.obj in Funds,
  ( m.id not in AuthorizedServices
    or (m.scope.obj not in Portfolio[useragent]) )
=> m is dropped
```

HiPoLDS rules are composed of two parts: the left part, before the '`=>`' construct, performs pattern matching on messages; the right part defines the security mechanisms that should apply – dropping the message, in this case. The `m:arep` clause applies when `m` is annotated with the `arep` information tag. Tags can be associated with a message based on either type inference and/or the structure of the message payload (we do not describe this here for brevity). `m.scope.obj` and `m.id` are extracted from the actual message content, part of which can also be identified from the type. In the example, `m.scope.obj` identifies the list of stocks in the fund E_Stock will be granted access to, and `m.id` refers to E_Stock. Finally, `Funds` and `AuthorizedServices` are sets, respectively comprising object identifiers on funds and external clients authorized to access fund data. `Portfolio` is a mapping between identifiers of user agents of employees (`useragent`) to the set of object identifiers for fund data.

4.3 Harnessing Types for Aspect-based Security

Scenario 3. Finally, we consider a scenario that requires some limited invasive modifications by OAuth stakeholders to the implementation of OAuth-related services. The OAuth standard mandates that sensitive data items e.g. authorization grants, tokens, and client credentials are stored securely. Developers frequently fail to adopt the best security mechanisms to protect assets, leading to vulnerable implementations. We consider a scenario in which security-relevant information, such as tokens, have to be stored at a remote user agent.

Solution. The OAuth standard contains a number of prescriptions that do not directly restrict the communication between stakeholders but instead manage security-relevant data has to be handled as part of service implementations. It restricts, for instance, how the user-agent's authenticated state (e.g., session cookie, HTML5 local storage) is to be stored (see OAuth standard, sec. 10.12). It prescribes that this data has to be kept in a location accessible only to the client and the user-agent. While this may be some common encrypted portion of the memory, it may also involve the use of special-purpose secured storage services, for instance, if the client uses a user agent remotely.

In order to ensure the use of a correct secure storage strategy, a combination of type-based security enforcement and aspect-based adaptation is used: the types that guide service discovery are extended in order to indicate the need of a particular storage strategy depending on the channel configuration between stakeholders and aspects are used to modify the implementation of services in order to use secure storage services if needed.

In the case of a remote user agent, the resource owner has to provide a typed channel for communication between the resource owner and the agent. Its type information makes explicit the need for a particular storage strategy. The strategy is then implemented using an aspect, by statically encapsulating the original token data structure in the user agent by a secure data structure. Alternatively, aspects can be used to dynamically encrypt the tokens and store them in a suitable data structure.

5 Related Work

We discuss related work in this section belonging to four domains: OAuth security issues, types for services, security policy languages, and service evolution, notably using aspects.

OAuth Security Issues. The main document describing OAuth 2.0 is [22] for a comprehensive OAuth security model and analysis. Several classes of attacks are discussed: key and secret storing and transmission, client authentication, token and refresh token, cross-site request forgery, guessing and phishing attacks, click-jacking, open redirectors and code injection. A real example of security problem with Twitter and OAuth 1.0a was described in [16]. The key and secret of a client can be discovered by a malicious third-party. The use of formal specification and verification techniques is a major approach to discover flaws in protocols. [12] uses the specification language Alloy and a SAT solver to discover security counterexample. In [4], the authors use the π-calculus, the WebSpy library and the ProVerif checker to make explicit various attacks on the OAuth protocol. They claim to find dozens of previously unknown vulnerabilities in connecting social networking like Twitter and Facebook with websites like Yahoo and WordPress.

Types for Services. The type system of [13] is based on a nested record type system with collections and universal polymorphism. This type system is neither recursive nor does it allow channel mobility; its checking algorithm is expensive even in this restricted setting. Sans and Cervesato [17] deal with an abstract model that covers code mobility which we do not address, we are only concerned with remote procedure calls. On the other hand, they consider functions rather than channels and they do not support sum types, nor recursive types. They require a centralized typing table collecting types of services published everywhere in the Internet and assume that this repository can be trusted. A distributed and typed π-calculus for mobile agents is described in [15]. The type system considers malicious agents with erroneous types. Type safety is enforced by dynamically type checking agents when they enter a site. In contrast to our work they do not consider channel discovery or subtyping. The last piece of work is [6] which applies semantics subtyping to the π-calculus. Despite the presence of a precise orchestration, their typing rules for services are similar to ours. We are not concerned with a precise process algebra for agent behaviors and this point is shared with all work on session types. But we focused on typing the communications between several agents leading to similar rules than in component systems (see [19]) and we also consider type attacks from malicious agents.

Security Policy Languages. With respect to security policy languages, HiPoLDS is particularly suited for this setting because it is especially de-

signed for complex distributed architectures, taking into account the fact that different security policies apply in different parts of the distributed system according to its security levels, and not all execution environments where services are running can be controlled. Notice that XACML [11] can be used in conjunction with OAuth 2.0 (e.g. for scope definitions), but it is not suited to describe the reference monitor behavior.

Similarly to HiPoLDS, Law Governed Interactions (LGI) [20] provides a hierarchical way of specifying the architecture of a distributed systems and security policies that apply only to a subset of such a system; policy enforcement is performed by reference monitors. Domains are governed by a mandatory policy, their law. However, the approach fails short to account for multiple stakeholders, since it does not consider that the enforcement might not always be possible - or at least not by an authority that is trusted enough to ensure the application of the law. Thus LGI requires that all reference monitors (running at any location of the distributed system) to be trusted by all participating entities: a strong assumption that cannot be applied in our scenario.

The same assumption is present in SPL [14], a language that like HiPoLDS allows to specify security requirements at different levels of abstraction. In addition to the requirement of trusting reference monitors, SPL is limited to access control policies, and does not allow specifying rules that results in reference monitors altering the messages that are passed between monitors, for example by encrypting content or by adding signatures or other type of security metadata.

The ConSpec language [1] aims at defining the behavior of reference monitors with a simple policy language similarly to our approach. This proposal focuses on the instrumentation of the control flow of an object-oriented program using before- and after- method modifiers. In contrast, our work aims at the high-level description of information flows, in particular materialized by the notions of messages, their types, and information tags. In our approach, a message can be mapped at the instrumentation phase to the interception of a protocol message at a client, a server, or an intermediate party, or to the inlining of a reference monitor controlling inter-component information flows at a protocol endpoint. Furthermore, ConSpec addresses neither the specification of multiple overlapping security policies, as illustrated in scenario 2 for instance, nor the definition of roles or groups.

Service Evolution and Aspects. Service evolution can be achieved in a flexible and non intrusive way using reference monitors [18] as long as

only the contents and recipients of messages have to be modified. Frequently, these changes are performed using dedicated monitors and reconfigurations of orchestrations, for instance, using aspects that modify BPEL-based service compositions (as, e.g., AO4BPEL [7]). Our approach to service evolution is novel in that our reference monitors are derived from HiPoLDS policy definitions and that our aspect system supports invasive modifications to service interceptors and implementations [9] that are required to resolve some security and interoperability issues of OAuth 2.0 (notably the scenario in Sec. 4.3).

6 Conclusion

The OAuth 2.0 protocol is an IETF standard already adopted by major internet application providers. However, it is often difficult to ensure that the implementation of authorization protocols are secure and interoperable because of the many optional features and different protocol flows of the OAuth framework. In this paper we use a type-based policy language in conjunction with reference monitors and aspect-oriented programming in order to tackle these issues. Types enable the precise definition of communicated data and the rigorous analysis of input data. Further, we integrate a policy language based on security domain and abstract rules to express security. Types and policies are implemented thanks to a reference monitor mechanism which encapsulates the agents that have to be adapted. For advanced evolutions that require invasive modifications of service interceptors and implementations, we use a new aspect-based system for service manipulation. Finally, we have shown three realistic evolution scenarios for which we have solved problems of input validation, interoperability and security issues. Future work is planned on the complete implementation of our framework on top of Apache's CXF web service model and its integration with RESTful service models.

References

1. Irem Aktug and Katsiaryna Naliuka. Conspec - a formal language for policy specification. *ENTCS*, 197(1):45 – 58, 2008. Proceedings of REM 2007.
2. Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. Well-Typed Services Cannot Go Wrong. Rapport de recherche RR-7899, INRIA, May 2012.
3. Ascola team. An aspect framework for CXF. `http://a4cloud.gforge.inria.fr/doku. php?id=start:aspect4cxf`, Jan. 2013.

4. Chetan Bansal, Karthikeyan Bhargavan, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF 2012, Cambridge, MA, USA*, pages 247–262. IEEE, 2012.
5. Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proc. of ICALP*, volume 3580 of *LNCS*, pages 30–34. Springer, 2005.
6. Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, May 2008.
7. Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *European Conf. on Web Services (ECOWS)*, volume 3250 of *LNCS*. Springer, 2004.
8. Omar Chebaro, Diana Allam, Hervé Grall, et al. Mechanisms for Property Preservation. Technical Report Deliverable D2.4, CESSA Project, July 2012.
9. Ronan-Alexandre Cherrueau, Omar Chebaro, and Mario Südholt. Flexible and expressive aspect-based control over service compositions in the cloud. In *4th Int. WS on Variability & Composition (VariComp)*. ACM DL, March 2013.
10. Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Oliveira, and Yves Roudier. Hipolds: A hierarchical security policy language for distributed systems. *Information Security Technical Report*, 2012.
11. OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. Technical report, OASIS, January 2013.
12. Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M. Pai, and Sanjay Singh. Formal verification of oauth 2.0 using alloy framework. In *CSNT '11*, pages 655–659, Washington, DC, USA, 2011. IEEE Computer Society.
13. Ken Q. Pu. Service description and analysis from a type theoretic approach. In *ICDE Workshops*, pages 379–386, 2007.
14. Carlos Ribeiro and Paulo Ferreira. A policy-oriented language for expressing security specifications. *International Journal of Network Security*, 5(3):299–316, 2007.
15. James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. *Journal of Automated Reasoning*, 31(3-4):335–370, 2003.
16. Paul Ryan. Compromising twitter's oauth security system. Technical report, Ars Technica, 2010.
17. Thierry Sans and Iliano Cervesato. QWeSST for type-safe web programming. 2010.
18. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
19. João Costa Seco and Luís Caires. A basic model of typed components. In *Proc. of ECOOP 2000*, LNCS 1850, pages 108–128. Springer, 2000.
20. Constantin Serban, W. Zhang, and N. Minsky. A decentralized mechanism for application level monitoring of distributed systems. In *Proceedings of CollaborateCom 2009*, pages 1–10. IEEE, 2009.
21. IETF Web Authorization (OAuth) Working Group. SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants. Technical Report V 17, Internet Engineering Task Force (IETF).
22. IETF Web Authorization (OAuth) Working Group. The OAuth 2.0 Authorization Framework. Technical Report RFC 6749, Internet Engineering Task Force (IETF), October 2012.
23. IETF Web Authorization (OAuth) Working Group. The OAuth 2.0 Authorization Framework: Bearer Token Usage. Technical Report RFC 6750, Internet Engineering Task Force (IETF), October 2012.

A Subtyping Rules and Endpoint Types Tables

Table 1 presents the main subtyping rules required in this paper. We

Table 1. Main Subtyping Rules

`A` $\leq$ `(A + B)` and `B` $\leq$ `(A + B)`
`(A + B)` $\leq$ `C` $\Longleftrightarrow$ `A` $\leq$ `C` and `B` $\leq$ `C`
`(A & B)` $\leq$ `A` and `(A & B)` $\leq$ `B`
`C` $\leq$ `(A & B)` $\Longleftrightarrow$ `C` $\leq$ `A` and `C` $\leq$ `B`
Covariance of record type `A` $\leq$ `A'` and `B` $\leq$ `B'` $\Longleftrightarrow$ `{"a":A; "b":B; ...}` $\leq$ `{"a":A'; "b":B'}`
Channel type contravariance `<R>` $\leq$ `<T>` $\Longleftrightarrow$ `T` $\leq$ `R`
Compatibility of a required service `r` connected to a provided one `p` `p:Provided` $\leq$ `r:Required`

give in Tables 2 the endpoint types (without the refresh token option) in ACF. In these tables `X.name` denotes a provided service named `name` from agent `X`. The notation `X.name (<numbering>)` corresponds to a required endpoint connected to `X.name`.

Table 2. ACF Provided and Required Endpoint Types Table

Provided Endpoint Types	
C.crep	`<({"grant":AuthCode; "state":State} + Token + DenyError) & Secure>`
AS.trep	`<(AuthCode` $\oplus$ `crep) & Secure>`
AS.arep	`<({"id":Credents ; "state":State}` $\oplus$ `Scope` $\oplus$ `C.crep) & Secure>`
Required Endpoint Types	
AS.crep (3)	`<({"grant":AuthCode; "state":State} + DenyError) & Secure>`
AS.crep (5)	`<(Token + DenyError) & Secure>`
C.trep (4)	`<(AuthCode` $\oplus$ `crep) & Secure>`
C.arep (1)	`<({"id":Credents ; "state":State}` $\oplus$ `Scope` $\oplus$ `C.crep) & Secure>`